Automatic phased mission system reliability model generation

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

• A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/13583](https://dspace.lboro.ac.uk/2134/13583)

Publisher: © Kathryn Sarah Stockwell

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
Automatic Phased Mission System
Reliability Model Generation

by

Kathryn Sarah Stockwell

Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy

of

Loughborough University

September 2013

© by Kathryn Sarah Stockwell 2013
Dedicated to my parents Marie and John
and to Tom
Abstract

There are many methods for modelling the reliability of systems based on component failure data. This task becomes more complex as systems increase in size, or undertake missions that comprise multiple discrete modes of operation, or phases. Existing techniques require certain levels of expertise in the model generation and calculation processes, meaning that risk and reliability assessments of systems can often be expensive and time-consuming. This is exacerbated as system complexity increases.

This thesis presents a novel method which generates reliability models for phased-mission systems, based on Petri nets, from simple input files. The process has been automated with a piece of software designed for engineers with little or no experience in the field of risk and reliability. The software can generate models for both repairable and non-repairable systems, allowing redundant components and maintenance cycles to be included in the model.

Further, the software includes a simulator for the generated models. This allows a user with simple input files to perform automatic model generation and simulation with a single piece of software, yielding detailed failure data on components, phases, missions and the overall system. A system can also be simulated across multiple consecutive missions. To assess performance, the software is compared with an analytical approach and found to match within ±5% in both the repairable and non-repairable cases.

The software documented in this thesis could serve as an aid to engineers designing new systems to validate the reliability of the system. This would not require specialist consultants or additional software, ensuring that the analysis provides results in a timely and cost-effective manner.

Keywords: Phased-Mission Systems, Automated, Model Generation, Simulation, Petri nets, Non-Repairable, Repairable, Operational Mode Tables and Decision Tables.
Acknowledgment

I would like to thank my supervisor, Sarah Dunnett, for her support throughout my Ph.D. I would also like to thank Chris, Liz and Tom, particularly for their friendship and office shenanigans.

I would also like to thank Lockheed Martin who have given me time to complete this Ph.D. and those particularly at Lockheed Martin who took the time to ensure I could achieve this goal.

I would lastly like to thank both the Stockwell and Offer families for their love, support and unending encouragement throughout the last 4 years. My parents who have always been there for me and Tom, for everything.
## Contents

List of Figures xiii

List of Tables xix

Principal Notation xxii

List of Acronyms xxiii

### 1 Introduction

1.1 Background 1

1.2 Research Objectives 2

1.3 Basic Definitions 3

1.3.1 Hazard Rate 3

1.3.2 Reliability and Unreliability 3

1.3.3 Availability and Unavailability 4

1.3.4 Maintenance Policies 4

1.3.5 Cut Sets and Minimal Cut Sets 6

1.3.6 Implicants and Prime Implicants 6

1.4 Reliability Techniques 6

1.4.1 Combinatorial 6

1.4.2 State-Space 19

1.4.3 Simulation 24

1.5 Summary 38

### 2 Phased-Mission Systems

2.1 Introduction 39

2.1.1 Types of phased-mission systems 40

2.1.2 Analytical Modelling Techniques 41

2.2 Non-Repairable Systems 41

2.2.1 Phase Fault Trees 41

2.2.2 Phase Modular Approach 54

2.2.3 Binary Decision Diagrams for Phased-Mission Systems 57

2.3 Repairable Systems 62

2.3.1 Markov applications in Phased-Mission Systems 62

2.3.2 System and Phase Petri Nets 69

2.4 Summary 71
3 Automated Techniques
3.1 Introduction .......................................................... 73
3.2 Methods for Automation of Reliability Models .................... 73
  3.2.1 Decision Table Methods ........................................ 73
  3.2.2 Digraph Method .................................................. 77
  3.2.3 Modified Decision Table method ................................ 77
  3.2.4 Cause-Consequence Diagrams ................................... 79
  3.2.5 Mini fault trees .................................................. 79
  3.2.6 Faultfinder ....................................................... 79
3.3 Summary .................................................................. 85

4 Modelling of Non-Repairable Systems ................................. 87
4.1 Introduction .............................................................. 87
4.2 Model Inputs ............................................................. 88
  4.2.1 Component Description ........................................... 88
  4.2.2 System Description ............................................... 91
  4.2.3 Circuit Description ............................................... 92
  4.2.4 Phase Description ................................................ 92
  4.2.5 Initial Conditions ................................................ 92
4.3 Petri Net Models ....................................................... 93
  4.3.1 Component Petri Nets ............................................ 93
  4.3.2 Circuit Petri Nets ................................ ................. 102
  4.3.3 System Petri Nets ................................................. 104
  4.3.4 Phase Petri Nets ................................................ 109
4.4 Algorithm ................................................................ 115
4.5 Summary ................................................................ 120

5 Application of the Procedure to Pressure Tank System .......... 123
5.1 Introduction .............................................................. 123
  5.1.1 The Pressure Tank System ...................................... 124
5.2 System and Mission Description ..................................... 125
  5.2.1 Components ....................................................... 125
  5.2.2 System Structure .................................................. 130
  5.2.3 Circuits .............................................................. 130
  5.2.4 Mission Profile ..................................................... 131
5.3 Pressure Tank System Model Construction ......................... 131
  5.3.1 Component and System Petri Nets ......................... 132
  5.3.2 Circuit Petri Nets ............................................... 132
  5.3.3 Phase Petri Net .................................................. 134
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5.1</td>
<td>File Input</td>
<td>221</td>
</tr>
<tr>
<td>7.5.2</td>
<td>System Storage</td>
<td>222</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Construction Procedure</td>
<td>222</td>
</tr>
<tr>
<td>7.6</td>
<td>Mission Abort</td>
<td>225</td>
</tr>
<tr>
<td>7.6.1</td>
<td>File Input</td>
<td>225</td>
</tr>
<tr>
<td>7.6.2</td>
<td>System Storage</td>
<td>225</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Construction Procedure</td>
<td>225</td>
</tr>
<tr>
<td>7.7</td>
<td>Simulating a Repairable System</td>
<td>229</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Simulation Algorithm</td>
<td>229</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Simulation of the model</td>
<td>230</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Simulating Transitions</td>
<td>230</td>
</tr>
<tr>
<td>7.8</td>
<td>Repairable Bulb System</td>
<td>230</td>
</tr>
<tr>
<td>7.8.1</td>
<td>Introduction</td>
<td>230</td>
</tr>
<tr>
<td>7.8.2</td>
<td>System Description</td>
<td>231</td>
</tr>
<tr>
<td>7.8.3</td>
<td>Mission Description</td>
<td>234</td>
</tr>
<tr>
<td>7.8.4</td>
<td>Maintenance Plan</td>
<td>234</td>
</tr>
<tr>
<td>7.8.5</td>
<td>Petri Net Models</td>
<td>234</td>
</tr>
<tr>
<td>7.8.6</td>
<td>Validation</td>
<td>237</td>
</tr>
<tr>
<td>7.9</td>
<td>Summary</td>
<td>242</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion and Further Work</td>
<td>243</td>
</tr>
<tr>
<td>8.1</td>
<td>Conclusion</td>
<td>243</td>
</tr>
<tr>
<td>8.2</td>
<td>Further Work</td>
<td>246</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Optimisation Study</td>
<td>246</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Minimal Cut Sets</td>
<td>246</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Automatic Generation of the System Structure File</td>
<td>247</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Multiple Interacting Systems</td>
<td>247</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Optimisation Study</td>
<td>246</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Minimal Cut Sets</td>
<td>246</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Automatic Generation of the System Structure File</td>
<td>247</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Multiple Interacting Systems</td>
<td>247</td>
</tr>
</tbody>
</table>

References | 249 |

A User Interaction | 255 |
| A.1 Menu Interaction | 255  |

B Pressure Tank System | 259 |
| B.1 Input Files | 259 |
| B.1.1 Project File | 259 |
| B.1.2 Component Files | 260 |
| B.1.3 System Structure File | 267 |
| B.1.4 Phase Transition Table File | 270 |
Contents

B.1.5 Simulation File ............................................. 271
B.1.6 Setup File .................................................. 273
B.2 Analytical Results ........................................... 273
  B.2.1 Single Mission .......................................... 273
B.3 Simulation Results ........................................... 273
  B.3.1 Single Mission .......................................... 273
  B.3.2 Multiple Missions ....................................... 273

C Bulb System .................................................... 279
  C.1 Input Files .................................................. 279
    C.1.1 Project File ........................................... 279
    C.1.2 Component Files ...................................... 280
    C.1.3 System Structure File ................................ 282
    C.1.4 Phase Transition Table File .......................... 283
    C.1.5 Simulation File ....................................... 284
    C.1.6 Setup File ............................................. 285
List of Figures

1.1 The bath-tub curve ............................................. 3
1.2 Example fault tree ........................................... 10
1.3 Example reliability block diagram ............................ 12
1.4 Reliability block diagram including series, parallel and voting systems ... 15
1.5 Reliability block diagram analysis steps .................... 15
1.6 Example binary decision diagram ............................. 16
1.7 Binary decision diagram illustrating $\text{ite}(X_1, f_1, f_2)$ ............ 17
1.8 Simple fault tree structure for conversion to a binary decision diagram ... 18
1.9 Markov model depicting a working, failed state system ............. 20
1.10 Two-component system ......................................... 25
1.11 Representation of direct sampling ............................. 26
1.12 Working and failed state system ............................... 29
1.13 Petri net with multiple transitions and multiple tokens in one place .... 30
1.14 Petri net ....................................................... 31
1.15 Petri net transition process ................................... 32
1.16 Petri net example and dual of the Petri net .................... 34
1.17 Petri nets illustrating absorption ............................... 36
1.18 Petri net and the equivalent reachability graph ................ 38
2.1 Phased mission of an aircraft flight .......................... 40
2.2 Process for a two phase mission to find the exact solution .......... 43
2.3 Generalised phase fault tree (La Band 2005) ..................... 47
2.4 Three phased-mission system .................................. 47
2.5 Phase fault tree construction of phase 2 ....................... 48
2.6 Phase fault tree construction of phase 3 ....................... 49
2.7 Extraction method for fault trees .............................. 49
2.8 Modularised fault tree ......................................... 55
2.9 System configuration for a three phase mission .................. 62
3.1 Operator-driven valve .......................................... 76
3.2 FAULTFINDER Structure (Hunt et al. 1993) .................. 80
4.1 Power Supply component ....................................... 88
4.2 Switch component ............................................. 89
4.3 Example of a simple system with one circuit ................... 91
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Example of a topology diagram</td>
</tr>
<tr>
<td>4.5</td>
<td>Procedure steps for the construction of a Petri net representing a power supply</td>
</tr>
<tr>
<td>4.6</td>
<td>Procedure steps for the construction of a Petri net representing a toggle switch using the Operational Mode Transition (OMT)</td>
</tr>
<tr>
<td>4.7</td>
<td>Procedure steps for the construction of a Petri net representing a toggle switch using the Decision Table (DT)</td>
</tr>
<tr>
<td>4.8</td>
<td>Example of using failure rates in Petri net models</td>
</tr>
<tr>
<td>4.9</td>
<td>Example of a component with multiple failure states and one operating mode</td>
</tr>
<tr>
<td>4.10</td>
<td>Example of a system with multiple circuits</td>
</tr>
<tr>
<td>4.11</td>
<td>Procedure steps for the construction of a Petri net representing circuit 1</td>
</tr>
<tr>
<td>4.12</td>
<td>Petri net for current and no current in circuit 2</td>
</tr>
<tr>
<td>4.13</td>
<td>Petri net for the monitoring of the state of circuit 2 and the connection to the System Petri Net (SPN)</td>
</tr>
<tr>
<td>4.14</td>
<td>Procedure steps for the construction of the system Petri net</td>
</tr>
<tr>
<td>4.15</td>
<td>System diagram and topology for a heater and fan system</td>
</tr>
<tr>
<td>4.16</td>
<td>Steps 1-5 of the construction procedure applied to the heater, fan system</td>
</tr>
<tr>
<td>4.17</td>
<td>Steps 6-8 of the construction procedure applied to the heater, fan system</td>
</tr>
<tr>
<td>4.18</td>
<td>Steps 9-10 of the construction procedure applied to the heater, fan system</td>
</tr>
<tr>
<td>4.19</td>
<td>Flow chart of the algorithm</td>
</tr>
<tr>
<td>5.1</td>
<td>Pressure Tank System</td>
</tr>
<tr>
<td>5.2</td>
<td>Schematic of Pressure Tank System</td>
</tr>
<tr>
<td>5.3</td>
<td>Component Petri Net (CPN) construction and integration of the component tables for the component push-switch (S1)</td>
</tr>
<tr>
<td>5.4</td>
<td>Component Petri net of the failure rate for components S1, CR, CT and V</td>
</tr>
<tr>
<td>5.5</td>
<td>Pressure Gauge, PG, transitions between the working state and the different failure states</td>
</tr>
<tr>
<td>5.6</td>
<td>Petri nets for components with multiple modes of operation</td>
</tr>
<tr>
<td>5.7</td>
<td>Petri nets for components within circuits</td>
</tr>
<tr>
<td>5.8</td>
<td>Component Petri nets for non-circuit components</td>
</tr>
<tr>
<td>5.9</td>
<td>Part one of the system PN</td>
</tr>
<tr>
<td>5.10</td>
<td>Part two of the system PN</td>
</tr>
<tr>
<td>5.11</td>
<td>Part three of the system PN</td>
</tr>
<tr>
<td>5.12</td>
<td>Part four of the system PN</td>
</tr>
<tr>
<td>5.13</td>
<td>Circuit Petri Nets for Circuits 1 to 4 of the Pressure Tank System</td>
</tr>
<tr>
<td>5.14</td>
<td>Circuit Petri Net for Circuit 5 of the Pressure Tank System</td>
</tr>
<tr>
<td>5.15</td>
<td>Circuit Petri net linkage Petri nets</td>
</tr>
<tr>
<td>5.16</td>
<td>Phase Petri net for the Pressure Tank System</td>
</tr>
</tbody>
</table>
List of Figures

6.1 Decision table file input format for single mode components .......... 151
6.2 Examples of the file format required for multiple mode components ... 152
6.3 Example topology file format ........................................... 154
6.4 Example mission file format .............................................. 155
6.5 Example simulation file format ........................................... 156
6.6 Class view of topSystem, fileHandler and compLib .................... 159
6.7 Class representation of the component class and its sub-classes ........ 160
6.8 First section of the algorithm for parsing and storing information within a .dt or .omt file ......................................................... 162
6.9 Second section of the algorithm for parsing and storing information within a .dt or .omt file ......................................................... 163
6.10 First section of the algorithm for parsing and storing information within a .ss file ............................................................. 164
6.11 Second section of the algorithm for parsing and storing information within a .ss file ............................................................. 165
6.12 Third section of the algorithm for parsing and storing information within a .ss file ............................................................. 166
6.13 First section of the algorithm for parsing and storing information within a .ptt file ............................................................. 168
6.14 Second section of the algorithm for parsing and storing information within a .ptt file ............................................................. 169
6.15 Third section of the algorithm for parsing and storing information within a .ptt file ............................................................. 170
6.16 Process for the generation of the circuit lists ................................ 172
6.17 Circuit detection method used for the circuit system in the pressure tank system ................................................................. 173
6.18 Flow chart of the process of the simulation of a transition ............... 184
6.19 Flow chart of the process of the simulation of the model, part 1 ......... 185
6.20 Flow chart section of the process of the simulation of the model, part 2 .... 186
6.21 Pressure tank system phase 1 fault tree .................................. 189
6.22 Pressure tank system phase 2 fault tree .................................. 189
6.23 Pressure tank system phase 3 fault tree .................................. 190
6.24 Pressure tank system phase 4 fault tree .................................. 190
6.25 Phase 2 simulation convergence results .................................. 192
6.26 Phase 3 simulation convergence results .................................. 193
6.27 Phase 4 simulation convergence results .................................. 193
6.28 Mission simulation convergence results .................................. 194
6.29 Section of the mission unreliability graph for a non-repairable pressure tank system ........................................ 195
6.30 Individual phase unavailability over time ........................... 196
6.31 Mission unavailability over time .................................... 197

7.1 Petri net showing the transition between working, failed and under repair states of a component ..................................... 200
7.2 General preventative Petri net ........................................ 201
7.3 File Input expressions within the simulation file ..................... 202
7.4 planDetails class view .................................................. 202
7.5 preventative class view ................................................ 202
7.6 maintenance class view ............................................... 203
7.7 Flow chart showing the steps to taking the file input and storing the preventative maintenance information ..................... 204
7.8 Flow chart showing the steps to taking the file input and storing the preventative maintenance information ..................... 207
7.9 corrective class view .................................................. 208
7.10 Flow chart for the process of taking in the file and interpreting the corrective maintenance plan ................................. 209
7.11 Petri net for a single mode component under a corrective maintenance plan ......................................................... 212
7.12 Petri net for multiple components maintained by a single maintenance engineer .................................................... 213
7.13 Petri net for a system-wide corrective maintenance plan with two maintenance engineers ............................................. 214
7.14 Declarations used within the STANDBY header within the simulation file ......................................................... 214
7.15 The standby class ........................................................ 215
7.16 Flow chart for the storing of the standby components ............. 216
7.17 Example of power supplies in cold standby ......................... 219
7.18 Example of work to fail to repair relationship for a single mode component in warm standby ...................................... 220
7.19 Example of power supplies in warm standby ....................... 220
7.20 Example of power supplies in hot standby ......................... 221
7.21 Example of a VOTING declaration .................................... 222
7.22 The voting class ........................................................ 222
7.23 Flow chart for the storing of the voting information .............. 223
7.24 An example of a 2-out-of-3 voting system Petri net ................ 224
7.25 File format for the ABORT header within the simulation file .... 225
7.26 The abortMission class ............................................... 226
7.27 The first part of the software algorithm to populate the system with abort conditions ........................................... 227
7.28 The second part of the software algorithm to populate the system with abort conditions ........................................... 228
7.29 Example representation of the abort process within a Phase Petri Net (PPN) ................................................ 228
7.30 Schematic of the bulb system ................................................ 231
7.31 System topology diagram for the bulb system ......................... 232
7.32 CPNs for the components of the bulb system ........................ 235
7.33 SPN of the bulb system ...................................................... 236
7.34 PPN of the bulb system ...................................................... 237
7.35 Markov Model of the Bulb System ....................................... 239
7.36 Simulation results for the repairable bulb system for 5,000 simulations ...... 242
A.1 Main Menu Screen ......................................................... 256
A.2 Main Menu Option 1 ....................................................... 256
A.3 Main Menu Option 3 ....................................................... 257
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Gate symbols (Andrews &amp; Moss 2002)</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>Event symbols (Andrews &amp; Moss 2002)</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Key to Petri nets</td>
<td>29</td>
</tr>
<tr>
<td>1.4</td>
<td>Fault tree symbols and Petri net equivalent</td>
<td>33</td>
</tr>
<tr>
<td>2.1</td>
<td>Algebraic law for phased-mission systems for $i &lt; j$</td>
<td>51</td>
</tr>
<tr>
<td>2.2</td>
<td>Phase algebra used by Zang et al. (1999) ($i &lt; j$)</td>
<td>57</td>
</tr>
<tr>
<td>2.3</td>
<td>States of component combinations for the example system</td>
<td>63</td>
</tr>
<tr>
<td>3.1</td>
<td>Complete decision table for component fuse</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>Reduced decision table for component fuse</td>
<td>74</td>
</tr>
<tr>
<td>3.3</td>
<td>Operator-driven valve state transition table</td>
<td>76</td>
</tr>
<tr>
<td>3.4</td>
<td>Operator-driven valve function table</td>
<td>76</td>
</tr>
<tr>
<td>3.5</td>
<td>Original decision table format for component Contact (Henry &amp; Andrews 1997)</td>
<td>78</td>
</tr>
<tr>
<td>3.6</td>
<td>Modified decision table format for component Contact (Henry &amp; Andrews 1997)</td>
<td>78</td>
</tr>
<tr>
<td>3.7</td>
<td>Example Decision Table for system description</td>
<td>82</td>
</tr>
<tr>
<td>4.1</td>
<td>Decision Table for a power supply</td>
<td>88</td>
</tr>
<tr>
<td>4.2</td>
<td>Decision Table for toggle switch</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>Operating mode table for toggle switch</td>
<td>89</td>
</tr>
<tr>
<td>4.4</td>
<td>Timer Relay Decision Table</td>
<td>91</td>
</tr>
<tr>
<td>4.5</td>
<td>Decision table for a pressure gauge</td>
<td>102</td>
</tr>
<tr>
<td>4.6</td>
<td>Phase Transition Table for heater fan system</td>
<td>112</td>
</tr>
<tr>
<td>4.7</td>
<td>Decision Table for Timer Relay TIM1</td>
<td>114</td>
</tr>
<tr>
<td>4.8</td>
<td>Decision Table for Timer Relays TIM2, TIM3</td>
<td>114</td>
</tr>
<tr>
<td>5.1</td>
<td>Operational mode table for push switch $S1$</td>
<td>126</td>
</tr>
<tr>
<td>5.2</td>
<td>Decision table for switches $S1$ and $S2$, and contacts $TC$ and $RC$</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>Operational mode table for toggle switch $S2$ and Valve $V$</td>
<td>126</td>
</tr>
<tr>
<td>5.4</td>
<td>Decision table for power supplies $PS1$ and $PS2$, and fuse $FS$</td>
<td>126</td>
</tr>
<tr>
<td>5.5</td>
<td>Decision table for relay $R$</td>
<td>127</td>
</tr>
<tr>
<td>5.6</td>
<td>Decision table for timer relay $TIM$</td>
<td>127</td>
</tr>
<tr>
<td>5.7</td>
<td>Operational mode table for timer relay contact $TC$ and relay contact $RC$</td>
<td>127</td>
</tr>
<tr>
<td>5.8</td>
<td>Decision table for junctions $J1$ and $J3$</td>
<td>127</td>
</tr>
<tr>
<td>Table Number</td>
<td>Table Title</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.9</td>
<td>Decision table for junctions $J_2$ and $J_4$</td>
<td>127</td>
</tr>
<tr>
<td>5.10</td>
<td>Decision table for motor $M$</td>
<td>128</td>
</tr>
<tr>
<td>5.11</td>
<td>Decision table for pump $P$</td>
<td>128</td>
</tr>
<tr>
<td>5.12</td>
<td>Decision table for tank $T$</td>
<td>128</td>
</tr>
<tr>
<td>5.13</td>
<td>Decision table for pressure gauge $PG$</td>
<td>128</td>
</tr>
<tr>
<td>5.14</td>
<td>Decision table for operator $OP$</td>
<td>129</td>
</tr>
<tr>
<td>5.15</td>
<td>Decision table for valve $V$</td>
<td>129</td>
</tr>
<tr>
<td>5.16</td>
<td>Pressure tank system component failure data</td>
<td>129</td>
</tr>
<tr>
<td>5.17</td>
<td>Phase transition table</td>
<td>132</td>
</tr>
<tr>
<td>6.1</td>
<td>Software arc type definitions</td>
<td>174</td>
</tr>
<tr>
<td>6.2</td>
<td>Pressure tank system component failure data</td>
<td>188</td>
</tr>
<tr>
<td>6.3</td>
<td>Pressure tank system simulation results from 10,000 simulations</td>
<td>192</td>
</tr>
<tr>
<td>7.1</td>
<td>Decision table for the component Bulb</td>
<td>232</td>
</tr>
<tr>
<td>7.2</td>
<td>Decision table for the Operator</td>
<td>233</td>
</tr>
<tr>
<td>7.3</td>
<td>Decision table for the component power supply</td>
<td>233</td>
</tr>
<tr>
<td>7.4</td>
<td>Decision table for the component Toggle Switch</td>
<td>233</td>
</tr>
<tr>
<td>7.5</td>
<td>Operational Mode Transition Table for the component toggle Switch</td>
<td>233</td>
</tr>
<tr>
<td>7.6</td>
<td>Failure and repair data for the components of the bulb system</td>
<td>234</td>
</tr>
<tr>
<td>7.7</td>
<td>Phase transition table for the bulb system</td>
<td>235</td>
</tr>
<tr>
<td>7.8</td>
<td>Markov model states for repairable bulb system</td>
<td>238</td>
</tr>
<tr>
<td>7.9</td>
<td>Bulb system simulation results for 5,000 simulations</td>
<td>241</td>
</tr>
<tr>
<td>7.10</td>
<td>Second set of simulation results for the bulb system for 5,000 simulations</td>
<td>241</td>
</tr>
<tr>
<td>B.1</td>
<td>Analytical values for the Unreliability and Reliability of each phase and each phase range</td>
<td>274</td>
</tr>
<tr>
<td>B.2</td>
<td>Simulation Results for the single mission condition for the Pressure Tank System (Simulations 0-4000)</td>
<td>275</td>
</tr>
<tr>
<td>B.3</td>
<td>Simulation Results for the single mission condition for the Pressure Tank System (Simulations 4050-8000)</td>
<td>276</td>
</tr>
<tr>
<td>B.4</td>
<td>Simulation Results for the single mission condition for the Pressure Tank System: (Simulations 8050-10000)</td>
<td>277</td>
</tr>
<tr>
<td>B.5</td>
<td>Analytical Results for the multiple mission condition for the Pressure Tank System</td>
<td>278</td>
</tr>
</tbody>
</table>
### Principal Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i$</td>
<td>failure of component $A$ in phase $i$</td>
</tr>
<tr>
<td>$\bar{A}_i$</td>
<td>success of component $A$ in phase $i$</td>
</tr>
<tr>
<td>$A_{ij}$</td>
<td>failure of component $A$ between phase $i$ and phase $j$</td>
</tr>
<tr>
<td>$\bar{A}_{ij}$</td>
<td>success of component $A$ between phase $i$ and phase $j$</td>
</tr>
<tr>
<td>$A(t)$</td>
<td>availability function</td>
</tr>
<tr>
<td>$[A]$</td>
<td>state transition matrix</td>
</tr>
<tr>
<td>$C$</td>
<td>consequence</td>
</tr>
<tr>
<td>$C_i$</td>
<td>existence of cut set $i$</td>
</tr>
<tr>
<td>$D$</td>
<td>vector of switching delays of transitions</td>
</tr>
<tr>
<td>$D_{t_i}$</td>
<td>transition number $i$, with a time to the transition $D$</td>
</tr>
<tr>
<td>$E$</td>
<td>set of arcs (edges) in a Petri net</td>
</tr>
<tr>
<td>$F(t)$</td>
<td>unreliability function</td>
</tr>
<tr>
<td>$G_i(t)$</td>
<td>Birnbaum’s measure of importance</td>
</tr>
<tr>
<td>$G_{ij}$</td>
<td>Birnbaum’s measure of importance for component $i$ in phase $j$</td>
</tr>
<tr>
<td>$G_{PN}$</td>
<td>set of generalised Petri net data</td>
</tr>
<tr>
<td>$h(t)$</td>
<td>hazard rate (conditional failure rate)</td>
</tr>
<tr>
<td>$I$</td>
<td>input(s)</td>
</tr>
<tr>
<td>$I_{CR_i}$</td>
<td>criticality measure of importance</td>
</tr>
<tr>
<td>$I_i^P$</td>
<td>phase importance of component $i$ in phase $j$</td>
</tr>
<tr>
<td>$I_{ij}^T$</td>
<td>transition importance of component $i$ in phase $j$</td>
</tr>
<tr>
<td>$M(0)$</td>
<td>initial marking</td>
</tr>
<tr>
<td>$N_C$</td>
<td>total number of cut sets</td>
</tr>
<tr>
<td>$O$</td>
<td>output(s)</td>
</tr>
<tr>
<td>$p(r_i)$</td>
<td>probability of the $i$th disjoint path to a terminal 1 node</td>
</tr>
<tr>
<td>$P$</td>
<td>probability</td>
</tr>
<tr>
<td>$P(C_i)$</td>
<td>probability of cut set $i$ occurring</td>
</tr>
<tr>
<td>$P_i$</td>
<td>place number $i$</td>
</tr>
<tr>
<td>$q_i$</td>
<td>unavailability of component $i$</td>
</tr>
<tr>
<td>$q_{ij}$</td>
<td>unavailability of component $i$ in phase $j$</td>
</tr>
<tr>
<td>$Q(t)$</td>
<td>unavailability function</td>
</tr>
<tr>
<td>$Q_j$</td>
<td>unavailability of phase $j$</td>
</tr>
<tr>
<td>$r_i$</td>
<td>reliability of component $i$</td>
</tr>
<tr>
<td>$r_{ij}$</td>
<td>reliability of component $i$ in phase $j$</td>
</tr>
<tr>
<td>$R$</td>
<td>risk</td>
</tr>
<tr>
<td>$R(t)$</td>
<td>reliability function</td>
</tr>
<tr>
<td>$R_j$</td>
<td>reliability of phase $j$</td>
</tr>
<tr>
<td>$S(T)$</td>
<td>probability of occupying state $S$</td>
</tr>
<tr>
<td>$t$</td>
<td>time</td>
</tr>
<tr>
<td>$t_i$</td>
<td>transition number $i$</td>
</tr>
<tr>
<td>$V_p$</td>
<td>set of places in a Petri net</td>
</tr>
</tbody>
</table>
$V_i$ set of transitions in a Petri net

$W$ weight of edges

**Greek**

$\theta$ test interval

$\lambda$ failure rate

$\mu$ mean time to failure

$\nu$ repair rate

$\tau$ mean repair time

**Subscripts**

AV average

E exact

LB lower bound

MCSUB minimal cut set upper bound

MISS mission

RE rare event

SYS system
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>Alternative Cause Stack</td>
</tr>
<tr>
<td>AFTC</td>
<td>Automated Fault Tree Construction</td>
</tr>
<tr>
<td>AFTCC</td>
<td>Automatic Fault Tree Construction Code</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CAT</td>
<td>Computer Automated Tree</td>
</tr>
<tr>
<td>CCD</td>
<td>Cause-Consequence Diagram</td>
</tr>
<tr>
<td>CCF</td>
<td>Common Cause Failure</td>
</tr>
<tr>
<td>CiPN</td>
<td>Circuit Petri Net</td>
</tr>
<tr>
<td>CPN</td>
<td>Component Petri Net</td>
</tr>
<tr>
<td>CSP</td>
<td>Cold Spare</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DSPN</td>
<td>Deterministic and Stochastic Petri Net</td>
</tr>
<tr>
<td>DT</td>
<td>Decision Table</td>
</tr>
<tr>
<td>FEHM</td>
<td>Fault/Error Handling Model</td>
</tr>
<tr>
<td>FORM</td>
<td>Fault-Occurrence/Repair Model</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>GPN</td>
<td>Generalised Petri Net</td>
</tr>
<tr>
<td>GSPN</td>
<td>Generalised Stochastic Petri Net</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphics User Interface</td>
</tr>
<tr>
<td>HSP</td>
<td>Hot Spare</td>
</tr>
<tr>
<td>IF-ITE</td>
<td>If-Then-Else</td>
</tr>
<tr>
<td>IN-EX</td>
<td>Inclusion-Exclusion</td>
</tr>
<tr>
<td>LB</td>
<td>Lower Bound</td>
</tr>
<tr>
<td>MBE</td>
<td>Module Basic Event</td>
</tr>
<tr>
<td>MCS</td>
<td>Minimal Cut Set</td>
</tr>
<tr>
<td>MCSUB</td>
<td>Minimal Cut Set Upper Bound</td>
</tr>
<tr>
<td>MPCT</td>
<td>Mission-Phase Change Times</td>
</tr>
<tr>
<td>MPN</td>
<td>Master Petri Net</td>
</tr>
<tr>
<td>MRP</td>
<td>Maintenance Recovery Period</td>
</tr>
<tr>
<td>MTBCF</td>
<td>Mean Time Between Critical Failures</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time to Failure</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time to Repair</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NFB</td>
<td>Negative Feedback</td>
</tr>
<tr>
<td>NFF</td>
<td>Negative Feedforward</td>
</tr>
<tr>
<td>OMT</td>
<td>Operational Mode Transition</td>
</tr>
<tr>
<td>PAND</td>
<td>Priority AND</td>
</tr>
<tr>
<td>PDO</td>
<td>Phased-Dependent Operation</td>
</tr>
<tr>
<td>PhN</td>
<td>Phase Net</td>
</tr>
<tr>
<td>PID</td>
<td>Piping and Instrumentation Diagram</td>
</tr>
<tr>
<td>PM</td>
<td>Phased-Mission</td>
</tr>
<tr>
<td>PMS</td>
<td>Phased-Mission System</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>PN</td>
<td>Petri Net</td>
</tr>
<tr>
<td>PPN</td>
<td>Phase Petri Net</td>
</tr>
<tr>
<td>RBD</td>
<td>Reliability Block Diagram</td>
</tr>
<tr>
<td>SDP</td>
<td>Sum of Disjoint Products</td>
</tr>
<tr>
<td>SPN</td>
<td>System Petri Net</td>
</tr>
</tbody>
</table>
1.1 Background

Risk and reliability has played a significant role in the design of major systems in a range of industries in recent decades. Assessing the reliability of systems has aided in improving the safety over the years. Major disasters such as the Chernobyl nuclear power plant in 1986 illustrate the necessity of robust risk and reliability analysis. System assessments applied at the design phase can reduce the chance of undesirable incidents occurring when a system is in operation. This can be achieved by identifying components or combinations of components within a system that could lead to an undesirable event; this information can then show design engineers the weaknesses in a design. The reliability of the design can then be enhanced, typically by introducing maintenance cycles or redundancy within the system.
Many methods have advanced since the Second World War to assess the probability (or frequency) of an undesirable, or hazardous, event occurring. The risk, \( R \), or *expected loss* can be defined as the product of the consequence, \( C \), and the probability, \( P \), of the undesirable event occurring. This is represented in equation 1.1.1.

\[
R = C \cdot P
\]  

(1.1.1)

If the risk is too high then the system is not suited to handle the undesirable event. By reducing one or both of the values that evaluate to the expected loss, an acceptable level of risk may be found. Consequence, for example, could be reduced by finding a way to reduce the number of people that work with the system. To reduce the probability of the undesirable event occurring, the system itself would need to change. This could be achieved through either a significant overhaul of the design or the introduction of redundancies and fail-safes. The reduction of this probability is the main focus of the work presented in this thesis; it introduces a more efficient means of acquiring the probability of the undesirable event occurring.

System reliability models are a way of representing the undesirable events and from them calculating the system reliability. They have been used over the years to determine whether a system is reliable to use for its intended purpose. These models can be used within the design phase to aid the designer in investigating implementation options. A reliability assessment of the design is generally required to prove it can perform to applicable standards. Usually a specialist team is required to complete this assessment, as the designers to do not have the necessary skills to complete this task. It can take a significant amount of time to generate the reliability models for the system; this can limit the scope for the analysis to influence the design.

### 1.2 Research Objectives

The work presented in this thesis provides a method by which to assess a system at the design phase allowing the reliability analysis of the system to influence the design in a timely fashion. The method uses information about the system and the mission the system is to undertake. This information is translated into a reliability model to assess the mission success/failure. The system information can be entered by a member of the design team rather than requiring specialist knowledge. The program then builds the model and gives the user the relevant data about the mission. This information can then be used to improve the system reliability.
1.3 Basic Definitions

This section gives a description of some of the commonly used terms in the thesis. The following information is based on Andrews and Moss (2002) which also contains further reading on these topics.

1.3.1 Hazard Rate

The hazard rate, or conditional failure rate, is a measure of the rate at which a component or system fails. By plotting the hazard rate against time, the curve of the graph usually follows that of the bath-tub curve. Figure 1.1 shows a generalised view of the bath-tub curve.

Figure 1.1 shows three distinct phases of a component or system’s life-cycle. The first, burn-in, shows a decreasing hazard rate as component manufacturing defects are most likely to present themselves early in the component’s life-cycle. The second phase, useful life, shows a constant failure rate as a result of random failures. The final phase, wearout, shows an increasing failure rate as the component/system deteriorates with age.

1.3.2 Reliability and Unreliability

The reliability, $R(t)$, of a component or system is the probability that an item (component, equipment, or system) will operate without failure for a stated period of time under specified conditions. This is a measure that the item under consideration is successful over a given period of time. Equation 1.3.1 represents the reliability of a system with a constant hazard rate, $\lambda$. Constant hazard rate is also referred to as the failure rate.
\[ R(t) = e^{-\lambda t} \]  

The unreliability, \( F(t) \), of a component or system is the probability that a component/system fails to work continuously over a specified time period, under specified conditions. The relationship between reliability and unreliability is given as follows:

\[ F(t) = 1 - R(t) \]

### 1.3.3 Availability and Unavailability

The availability, \( A(t) \), of a component or system is defined as the probability that the component or system is working at a particular instant. Alternatively, it is the fraction of the total time the component or system is able to undertake its required function. The availability of a system is an important measure of the performance of the system. This value is calculated when a system failure can be tolerated and repair can be initiated. Equation 1.3.3 represents the availability of a system.

\[ A = \frac{MTTF}{MTTF + MTTR} \]

Where the Mean Time to Failure (MTTF) is defined as the reciprocal of the failure rate \( \left( \frac{1}{\lambda} \right) \) and the Mean Time to Repair (MTTR) is defined as the average time taken from the failure of a system to its start-up, \( \tau \). The MTTR is also defined as the reciprocal of the repair rate \( \left( \frac{1}{\nu} \right) \).

The unavailability, \( Q(t) \), of a component or system is the counterpart to availability and is the probability that a component or system does not perform its required function for time \( t \). Unavailability has the following relationship:

\[ Q(t) = 1 - A(t) \]

### 1.3.4 Maintenance Policies

There are three types of maintenance policies for systems or components: no repair, scheduled maintenance and unscheduled maintenance. Each of these is given in detail below.

#### 1.3.4.1 No Repair

For this type of policy there is no maintenance once a system fails. If a system is said to be working at a given time \( t \), then the system must have been working continuously up
to time $t$. Therefore the reliability and availability are equal. Equation 1.3.1 would be applicable for analysing a system with this policy.

### 1.3.4.2 Scheduled Maintenance

Faults in systems do not always become apparent the moment they have failed. This can occur when a system is dormant and can only be detected when there is a demand on the system, or is discovered during a scheduled maintenance. This can be quantified by finding the probability of the system being in a failed state at any time by equation 1.3.5.

$$Q_{AV} = \lambda \left( \frac{\theta}{2} + \tau \right) \quad (1.3.5)$$

Where $\lambda$ is the unrevealed failure rate of the system and $\theta$ is the test interval.

Equation 1.3.5 can be approximated to equation 1.3.6 when the mean repair time, $\tau$, is much shorter than the test interval, $\theta$.

$$Q_{AV} = \frac{\lambda \theta}{2} \quad (1.3.6)$$

For the scheduled maintenance policy an inspection is carried out after a fixed time interval. When a failure is discovered during this inspection, repair is initiated. As this maintenance is based on the time between inspections, $\theta$, the unavailability is a function of this time. Therefore, equation 1.3.5 is used for the average unavailability of the system. Another form of the average unavailability can be found from integrating between the interval times as shown below. This is more accurate than the simplified equation given in equation 1.3.6. Equation 1.3.7 shows the integral between $t = 0$ and $t = \theta$, the first inspection period. This equation represents the unavailability of a system (or component). Between these inspection intervals the system (or component) is non-repairable.

$$Q_{AV} = \frac{1}{\theta} \int_0^\theta 1 - e^{-\lambda t} dt \quad (1.3.7)$$

By integrating, equation 1.3.7 this gives equation 1.3.8.

$$Q_{AV} = 1 - \frac{1}{\lambda \theta} \left( 1 - e^{-\lambda \theta} \right) \quad (1.3.8)$$

### 1.3.4.3 Unscheduled Maintenance

This policy initiates any repairs when a failure occurs. For this type of maintenance policy the analysis is only dependent on the failure and repair rate of the system (or component), as the fault is known as soon as it occurs; therefore there is no detection time. By the use of Laplace transforms it can be shown that the unavailability of a system (or component) is given by equation 1.3.9.
\[ Q(t) = \frac{\lambda}{\lambda + \nu} [1 - e^{-(\lambda+\nu)t}] \] 

(1.3.9)

For components that have settled down into their steady state, taking \( t \to \infty \) in equation 1.3.9 gives the steady state equation, equation 1.3.10.

\[ Q = \frac{\lambda}{\lambda + \nu} \] 

(1.3.10)

This can be simplified further, given that the MTTF will be significantly larger than the MTTR, therefore reducing equation 1.3.10 to equation 1.3.11.

\[ Q = \lambda \tau \] 

(1.3.11)

### 1.3.5 Cut Sets and Minimal Cut Sets

A failure mode, or system failure mode, is the failure of a system that can occur through the failure of a single component or a combination of components in that system. These failure modes can be defined by cut sets. Cut sets are a list of basic components or combinations of basic components that, should they fail, would cause a system failure event. Minimal cut sets are an extension of this concept that expresses the minimal set of components that is sufficient to cause each failure event.

### 1.3.6 Implicants and Prime Implicants

Implicants and Prime Implicants are similar to cut sets and minimal cut sets, in that they show what components, or combination of components, cause a system failure event. The difference is that implicants are combinations of working and failed components that cause a system failure event. Prime implicants are the minimal, but sufficient combinations of working and failed components required to cause a failure event.

### 1.4 Reliability Techniques

As the work focuses on the automatic building of a reliability model, the chosen modelling technique should cater for all types of systems and missions. The following section discusses the different modelling techniques available.

#### 1.4.1 Combinatorial

The following techniques look at the combinations of failure, usually of components, and the effects they have on the overall system state.
1.4. Reliability Techniques

1.4.1.1 Fault Tree Analysis

H. A. Watson of Bell Laboratories in 1961 whilst connected to the US Air Force contract to study the Minuteman Launch control system first conceived the idea of Fault Tree Analysis (FTA). D. Haasl of Boeing Company saw the merits and the value of this technique, and with a team, used this method for the whole Minuteman study. From this point onwards the Boeing Company used FTA for the design of commercial aircraft. Boeing Company in conjunction with the University of Washington in 1965 sponsored the first System Safety Conference where FTA papers were first presented. The interest of FTA soon spread beyond the Aerospace industry, with particular interest shown in the Nuclear industry (Ericson II 1999).

Fault tree analysis is a method of graphically displaying how an undesirable event could occur. This is accomplished using different symbols to demonstrate how components are linked to other components in the event of an undesirable event. From this structure the probability of system failure can be calculated.

When constructing a fault tree, the top event must first be described. The top event is the resulting failure of a system or process, i.e. a specific system failure mode. For example a top event could be “Landing gear failure”. From the top event branches are constructed which are used to show how such an event could occur. To construct the branches of the fault tree a number of gates and events are used to link them. These are listed in Table 1.1 and Table 1.2 respectively. For the full list of gate and event symbols see Andrews and Moss (2002), which also provides background information on this section.

Qualitative Analysis

Qualitative analysis of a fault tree is used to identify the failure modes of a system. These failure modes are used to quantify the potential failure of a system. These failure modes are defined as cut sets. These are discussed in Section 1.3.5.

To find the cut sets of a fault tree, there are two main approaches that can be taken. The top-down and the bottom-up approach. Logic expressions are used in conjunction with three laws in order to obtain the cut sets. In these expressions; ‘AND’ is represented by ‘.’ and ‘OR’ by ‘+’. The following are the three laws used in the process;

1. **Distributive Law**: This is used to expand out any brackets, e.g. \((A + B)(C + D) = A.C + A.D + B.C + B.D\)

2. **Idempotent Law**: This is used to remove repeated cut sets, e.g. \(A + A = A\), and is used to remove repeated failure events, e.g. \(A.A = A\)

3. **Absorption Law**: This is used to remove unnecessary non-minimal combinations, e.g. \(A + A.C = A\)
Table 1.1: Gate symbols (Andrews & Moss 2002)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="AND gate" /></td>
<td>AND gate</td>
<td>Output event occurs if all input events occur simultaneously.</td>
</tr>
<tr>
<td><img src="image2" alt="OR gate" /></td>
<td>OR gate</td>
<td>Output event occurs if at least one of the input events occur.</td>
</tr>
<tr>
<td><img src="image3" alt="k-out-of-n gate" /></td>
<td>k-out-of-n gate</td>
<td>Output event occurs if at least k events out of n events occur. E.g. 2-out-of-3.</td>
</tr>
<tr>
<td><img src="image4" alt="NOT gate" /></td>
<td>NOT gate</td>
<td>Output event occurs if the input events DO NOT occur.</td>
</tr>
</tbody>
</table>

To demonstrate each of these methods an example of a fault tree was constructed for a system with six basic events A, B, C, D and E which is given in Figure 1.2.

The top-down approach begins with the top event and is expanded in terms of the next level gate. In the case of the fault tree in Figure 1.2 this would be the AND gate. The first expansion would therefore be the inputs into this gate, \( G_1 \cdot G_2 \). From there the next level gate would be considered for both \( G_1 \) and \( G_2 \), continuously expanding the next level gate until the top event is presented in terms of only the basic events. Equation 1.4.1 shows the top event for the fault tree in Figure 1.2.

\[
T = A.C.D + A.C.E + A.B.C.F + B.D + B.D.E + B.D.F + D.E + E + B.E.F \tag{1.4.1}
\]

Hence the minimal cut sets are:

\[
T = A.C.D + A.B.C.F + B.D + E \tag{1.4.2}
\]

The bottom-up approach is similar to the top-down approach, but instead of starting from the top event, this approach moves from the basic events up through the fault tree. By using the three laws given above, the top event in terms of the minimal cuts are obtained. This leads to the same expression as equation 1.4.2.

Any of these methods can be used to obtain the top event; the choice of which one to
Table 1.2: Event symbols (Andrews & Moss 2002)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top or intermediate event</td>
<td>Describes the event occurring such as the top event or the intermediate steps after the top event. These are connected to logic gates.</td>
</tr>
<tr>
<td></td>
<td>Basic event</td>
<td>These are found at the end of the branches of a fault tree. These usually represent the basic components of a system.</td>
</tr>
<tr>
<td></td>
<td>Transfer symbol</td>
<td>Represents a part of the fault tree that is repeated elsewhere.</td>
</tr>
</tbody>
</table>

use is entirely up to the individual as each method provides the same result. Once the minimal cut sets are obtained the fault tree can then be analysed quantitatively.

Quantitative Analysis

Quantitative analysis of fault trees can determine the performance of the overall system, as it can be used to determine the probability, or frequency, of the particular system failure mode under consideration. Two aspects are discussed here: the top event probability and importance measures.

**Top Event Failure Probability**

The top event failure probability is dependent on what the top event is and is calculated by combining the failure probability of each minimal cut set. This can be achieved using the **inclusion-exclusion principle**. The probability of a minimal cut set occurring is the product of the probability of each component in the cut set occurring. The top event occurs if any one of the minimal cut sets occurs, so the unavailability of the system can be represented by equation 1.4.3.

\[
Q_E = \sum_{i=1}^{N_C} P(C_i) - \sum_{i=2}^{N_C} \sum_{j=1}^{i-1} P(C_i \cap C_j) + \cdots + (-1)^{N_C-1} P(C_1 \cap C_2 \cap \cdots \cap C_{N_C}) \quad (1.4.3)
\]

Where \( Q_E \) is the exact probability of the top event, \( N_C \) is the total number of cut sets and \( C_i \) and \( C_j \) are the \( i \)th and the \( j \)th minimal cut set.

For large systems, many minimal cut sets will exist that could cause a top event failure,
hence using the inclusion-exclusion principle can be time consuming and computationally expensive. Approximations are therefore considered to quantify the top event probability. The first of the approximations is the rare event approximation or upper bound. This approximation takes only the first term of the inclusion-exclusion equation, as shown in equation 1.4.4.

\[ Q_{UB} = \sum_{i=1}^{N_C} P(C_i) \]  

(1.4.4)

The next approximation is the lower bound, which takes the first and second term of equation 1.4.3 as shown in equation 1.4.5.

\[ Q_{LB} = \sum_{i=1}^{N_C} P(C_i) - \sum_{i=2}^{N_C} \sum_{j=1}^{i-1} P(C_i \cap C_j) \]  

(1.4.5)

The final approximation considered here is the Minimal Cut Set Upper Bound (MCSUB), which is described using equation 1.4.6.

\[ Q_{MCSUB} = 1 - \prod_{i=1}^{N_C} (1 - P(C_i)) \]  

(1.4.6)

The relationship between these approximations is given in equation 1.4.7.

\[ Q_{LB} \leq Q_E \leq Q_{MCSUB} \leq Q_{UB} \]  

(1.4.7)
1.4. Reliability Techniques

**Importance Measures**

In system reliability analysis, importance measures can be calculated for each component or minimal cut set and are used to identify weak areas of a system that could lead to, or be a contributing factor leading to, a top event. The measures assign a numerical value to each basic event or minimal cut set which allows them to be ranked in order of their contribution to the occurrence to the top event. There are many importance measures available for system reliability (see Andrews and Moss ((2002))). Two of the most common are Birnbaum’s measure of importance and the criticality measure of importance (Birnbaum (1969)). These measures are both based on a critical system state. The critical system state of component \( i \) is the case in which a failure in component \( i \) causes a failure of the entire system. Birnbaum’s measure of importance, \( G_i \), is defined as the probability that the system is in a critical state for component \( i \). Equation 1.4.8 is Birnbaum’s measure of importance.

\[
G_i(t) = \frac{\partial Q_{SYS}(t)}{\partial q_i(t)}
\]  

(1.4.8)

Where \( q_i(t) \) is the probability of failure of component \( i \) and \( Q_{SYS}(t) \) is the probability of failure of the system.

Birnbaum’s importance measure was built on to include the contribution from component \( i \) to the system failure; this is the criticality measure of importance. Equation 1.4.9 gives the criticality measure as the proportion that component \( i \) failures could cause the system to fail.

\[
I_{CR_i} = \frac{G_i(t)q_i(t)}{Q_{SYS}(t)}
\]  

(1.4.9)

1.4.1.2 Reliability Block Diagram

Reliability Block Diagrams (RBDs) or Reliability Networks are a method of determining system reliability using block diagrams to show system structure. Unlike fault trees, RBDs are success orientated and so the dependencies between the components represent how the system will function. (Andrews & Moss 2002).

RBDs consist of the following five features; a start node, an end node, a set of nodes, \( V \), a set of edges, \( E \), and an incidence function, \( \phi \). The incidence function is used to associate each edge with a set of ordered nodes. The edges are used to represent the components within a given system. The nodes are used to show the structure of the system, in that they are the points at which components are joined. An example RBD is given in Figure 1.3 constructed from four components, with four links between them. Each of the features given above can be defined as following for the example system:
To show that a component is working in the system the edges $X \rightarrow (i, j)$ represent that there is a path from node $i$ to node $j$ if component $X$ is working normally. The path is broken if component $X$ fails. Using the RBD in Figure 1.3 there are two paths that can be taken from the start node, 1 to the end node 4. If a path exists between these two nodes then the system is in a working state.

**Series Reliability Block Diagrams**

When a system is non-redundant i.e. it will not tolerate failures, then the components are placed in series, dictating that if any component in the system fails then the system fails. The general solution for $n$ components for the unreliability and reliability of a series RBD are given in equations 1.4.13 and 1.4.14, respectively. When evaluating the reliability of a system, the greater the number of components, the less reliable the system is.

$$Q_{SYS} = 1 - \prod_{i=1}^{n} (1 - q_i)$$
Parallel Reliability Block Diagrams

In comparison to series block diagrams where all components must be working for a system to work, parallel RBDs require only one path from the start node to the end node to exist for the system to work. The general form for \( n \) components for the unreliability and reliability of a system with parallel components are given in equations 1.4.15 and 1.4.16, respectively. The greater the number of components the lower the probability of failure.

\[
R_{SYS} = \prod_{i=1}^{n} r_i \quad (1.4.14)
\]

\[
Q_{SYS} = \prod_{i=1}^{n} q_i \quad (1.4.15)
\]

\[
R_{SYS} = 1 - \prod_{i=1}^{n}(1 - r_i) \quad (1.4.16)
\]

RBDs can also be made from both series and parallel connections, such as Figure 1.3. If within a branch of a parallel section exists a series of components, then the series of components are assessed first using equations 1.4.13 and 1.4.14. These series components can then be considered as a single component, as the parallel components are assessed using equations 1.4.15 and 1.4.16. If there were two sets of parallel components in series, then each set of parallel components would be assessed first and each set treated as a single component and then these two single components would be assessed in series.

Voting Systems in Reliability Block Diagrams

Voting systems are used when a system requires a combination of \( k \)-out-of-\( n \) components to be in a working state, where \( n \) represents the number of components in the voting system, of which \( k \) must work for the system to function. Sometimes the components within a voting system are identical and therefore are used in a redundant capacity. For analysis purposes to begin with, this will be assumed. The value of \( k \) and \( n \) are given at the side of the voting system in the form of a fraction, for example, where \( k \) is 2 and \( n \) is 3, the voting system would be represented by \( \frac{2}{3} \) at the side of the parallel RBD. The general form for the calculation of the reliability for each combination of working components is given in equation 1.4.17.

\[
P(k \text{ components work}) = \binom{n}{k} r^k q^{n-k} \quad (1.4.17)
\]

The unreliability of the voting system can also be calculated in a similar way with \( r \) and \( q \) reversed in equation 1.4.17. Using the example of 2-out-of-3 must work for the voting
system to be successful, equation 1.4.18 shows the unreliability of the voting system. This shows the number of components that, should they fail, would cause the system to fail.

\[
Q_{SYS} = P(\text{Two components fail}) + P(\text{Three components fail}) \\
= 3q^2r + q^3
\]  

(1.4.18)

When a voting system has non-identical components, then the analysis reflects this by producing each combination of working and failed components required to fail the system. So in the case of 2-out-of-3, there would be 3 combinations which consisted of 2 components failed and 1 working and 1 combination where all 3 components have failed.

Combined series, parallel and voting Reliability Block Diagrams

There are many ways in which series, parallel and voting systems can be incorporated into a RBD. To analyse such systems the RBD would be broken down into stages. Voting systems would first be assessed and this would then be followed by any components in series and then in parallel. For example the RBD given in Figure 1.4, the voting system would be analysed first. In this system the components within the voting system are identical components, therefore the reliability and unreliability of each component is \( r \) and \( q \), respectively. The analysis of the voting system can be seen in equations 1.4.19 and 1.4.20.

\[
q_1 = 3q^2r + q^3
\]  

(1.4.19)

\[
r_1 = 3r^2q + r^3
\]  

(1.4.20)

With the analysis of the voting system complete, this can become a single component within the RBD as seen in Figure 1.5a denoted 1. With no other voting systems present, the analysis of the system moves on to the two components in series, \( A \) and \( B \). The analysis of the series components is given in equations 1.4.21 and 1.4.22.

\[
q_2 = 1 - (1 - q_A)(1 - q_B)
\]  

(1.4.21)

\[
r_2 = r_A r_B
\]  

(1.4.22)

The series components, \( A \) and \( B \), are now represented by component 2, as seen in Figure 1.5b. The parallel components, \( C \) and 2 can now be analysed. The analysis of the parallel components can be seen in equations 1.4.23 and 1.4.24.
1.4. Reliability Techniques

Figure 1.4: Reliability block diagram including series, parallel and voting systems

(a) Voting system components represented by block component 1
(b) Series components, A and B, represented by block component 2
(c) Parallel components, 1 and C, represented by block component 3
(d) Series components, 1 and 3, represented by block component SYS

Figure 1.5: Reliability block diagram analysis steps

\[ q_3 = q_2 q_C \]  \hspace{1cm} (1.4.23)

\[ r_3 = 1 - (1 - r_2)(1 - r_C) \]  \hspace{1cm} (1.4.24)

The parallel components are now represented by component 3, as seen in Figure 1.5c. With the system now consisting of two components in series, components 3 and 1, these can be analysed to find the reliability and unreliability of the system. The result of which can be seen in equation 1.4.25 and 1.4.26.

\[ Q_{SYS} = q_3 + q_1 - q_3 q_1 \]  \hspace{1cm} (1.4.25)

\[ R_{SYS} = r_3 r_1 \]  \hspace{1cm} (1.4.26)
1.4.1.3 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) were first introduced within the field of reliability by Rauzy (1993). The method was introduced to aid in industrial scale fault tree analysis. The task of generating the minimal cut sets can be computationally heavy when handling large industrial systems. The algorithm proposed in this paper was to increase the efficiency of handling the minimal cut sets. This will be discussed in detail later in this section. First, this section covers the basic structure of a BDD.

A BDD has root, terminal and non-terminal nodes (also known as vertices) that are connected by branches. These Directed Acyclic Graphs (DAGs) only follow one direction and therefore do not loop back at any point. A root node is the start of the BDD and represents a basic event of a fault tree, it always has two branches connected underneath it. The left branch is the failure or occurrence of the basic event, denoted ‘1’. The right branch is the success of the basic event, component working, and denoted ‘0’. Each of these branches either becomes a terminal node or is connected to another basic event. The terminal nodes of a BDD represent the final state of the system, denoted by a ‘0’ representing a working system state and a ‘1’ representing a failed system state. The non-terminal nodes represent the basic events, these too always have two branches connected underneath it. An example of a BDD is given in Figure 1.6. The terminal and non-terminal nodes are labelled on the diagram. The BDD size is expressed as the number of non-terminal terms.

The paths of the BDD always begin with the root node. Each path moves through non-terminal nodes, until a terminal node is found. If a ‘1’ terminal node is found then this signifies a cut set. For example taking the BDD in Figure 1.6, this has two ‘1’ terminal node paths; \( A \) and \( \overline{A}, B, C \). Ignoring the success of basic event \( A \) in the second path, the cut sets are found as \{ \( A \) \}, \{ \( B, C \) \}.

As the ordering is very important in a BDD, the basic events need to be considered in
an order so that the system BDD can be constructed in an efficient manner; otherwise the BDD would become very large and computationally inefficient.

If-Then-Else Structure

Rauzy (1993) describes a method in which to form a BDD from a fault tree using a technique referred to as If-Then-Else (ite). The idea was to represent the gates of a fault tree using ite. Taking the top event to be expressed as the Boolean function, \( f(X) \) and pivoting about Boolean variable \( X_1 \), Shannon’s expansion is expressed as equation 1.4.27.

In equation 1.4.27 \( f_1 \) and \( f_2 \) are functions \( f(X) \) with \( X_1 = 1 \) and \( X_1 = 0 \), respectively.

\[
f(X) = X_1 \cdot f_1 + \overline{X_1} \cdot f_2 \quad (1.4.27)
\]

\( \text{ite}(X_1, f_1, f_2) \) represents the structure given in equation 1.4.27. This states that if \( X_1 \) fails, then consider \( f_1 \), else consider \( f_2 \). The BDD for this is shown in Figure 1.7.

To construct a BDD from a fault tree each basic event \( x \) is given the structure \( \text{ite}(x, 1, 0) \), this forms the basis of the full BDD. The tree is then considered from the bottom up and rules that are applied to connect basic and intermediate events are given below (where \( X \) and \( Y \) are variables):

If \( X < Y \) (i.e. \( X \) is considered before \( Y \))

\[
J \oplus H = \text{ite}(X, f_1 \oplus H, f_2 \oplus H) \quad (1.4.28)
\]

If \( X = Y \)

\[
J \oplus H = \text{ite}(X, f_1 \oplus g_1, f_2 \oplus g_2) \quad (1.4.29)
\]

Where equations 1.4.30 and 1.4.31 are the gate inputs.

\[
J = \text{ite}(X, f_1, f_2) \quad (1.4.30)
\]
Figure 1.8: Simple fault tree structure for conversion to a binary decision diagram

\[ H = \text{ite}(Y, g_1, g_2) \]  

(1.4.31)

Using this \text{ite} technique the repetition of nodes is avoided. An example of this technique in use is shown for the fault tree given in Figure 1.8.

With the ordering \( A < B < C \);

\[
G_1 = B.C = \text{ite}(B, 1, 0).\text{ite}(C, 1, 0) = \text{ite}(B, 1, \text{ite}(C, 1, 0), 0, \text{ite}(C, 1, 0)) = \text{ite}(B, \text{ite}(C, 1, 0), 0) 
\]  

(1.4.32)

\[
T = A + B.C = \text{ite}(A, 1, 0) + \text{ite}(B, \text{ite}(C, 1, 0), 0) = \text{ite}(A, 1 + \text{ite}(B, \text{ite}(C, 1, 0), 0), 0 + \text{ite}(B, \text{ite}(C, 1, 0), 0)) = \text{ite}(A, 1, \text{ite}(B, \text{ite}(C, 1, 0), 0)) 
\]  

(1.4.33)

This produces the BDD shown in Figure 1.6.

Rauzy (1993) also describes an algorithm for fault tree analysis which uses BDDs for the purpose of fault tree management. The purpose of the algorithm was to increase the efficiency at which the fault tree minimal cut sets and probability of failures could be found. The idea was to obtain one BDD from a fault tree and then apply a minimisation process.
to it to obtain a minimal BDD which represents the minimal cut sets of the fault tree. The minimal cut sets are found using the paths of the minimal BDD. The probability of the root event is found using Shannon’s decomposition and the terminal events of the BDD.

Reay and Andrews (2002) describe an efficient method to convert fault trees to BDDs. To simplify the fault tree before conversion, the FAUNET reduction (discussed later in section 2.2.1.2) covers the techniques of contraction, factorisation and extraction. Once these techniques have been employed, common structures (i.e. arrangements of gates and branches) are identified as modules within the fault tree. These modules contain no basic events that occur elsewhere in the fault tree. This makes analysing the whole fault tree easier by analysing each of the modules first and then substituting these into the higher-level fault tree. Once all the modules are identified these can be converted into BDDs. As the modules all have different properties, a different ordering system is chosen for each module, in order to take into account each modules’ individual properties.

Quantification of Binary Decision Diagrams

As each path of a BDD to a terminal node ‘1’ is disjoint, the top event probability, $Q$ is given in equation 1.4.34.

$$ Q = \sum_{i=1}^{n} p(r_i) \quad (1.4.34) $$

Where $p(r_i)$ is the probability of the $i$th disjoint path to a terminal 1 node.

1.4.2 State-Space

1.4.2.1 Markov Analysis

The first published work on Markov chains was by Andrei A. Markov in 1906, this was the start of much study of stochastic processes with many applications (Ching & Ng 2006). Markov models are used when dependencies exist between basic events and when failure rates do not vary with time. Fault trees cannot be used in this case as they cannot model statistical dependencies between components, i.e. the failure of one component cannot affect the failure of another.

There are two different types of Markov models to consider; the first is Discrete Markov chains and the second is Continuous Markov processes. These can both be defined in terms of time and space. Discrete systems move from one state to another at set points in time, whereas continuous systems move from one state to another at any point in time. Discrete systems have a set of non-overlapping exhaustive states identified, where the system must be in one of these at any given time. Continuous system states can degrade continuously between working and failed.
Markov models consist of two basic components; states and transitions. States are representative of the state in which the component resides, for example, working, failed or standby. Transitions show either a failure or a repair event between the states. Figure 1.9 gives an example of a simple working, failed state system.

- States:
  - State 1: Working
  - State 2: Failed

- Transitions:
  - $P($Failure Transition$)$: from state 1 to state 2 denoted $\lambda \ dt$
  - $P($Repair Transition$)$: from state 2 to state 1 denoted $\nu \ dt$

Where $\lambda$ and $\nu$ are the failure rate and the repair rate, respectively.

Using these models the probability of being in any of the states of a system can be calculated. These calculations will be given in detail later.

The events that are considered dependent in reliability modelling can include standby redundancy, common-cause and failure/repair processes. Standby redundancy incorporates a standby component that is brought into effect should the primary component fail, or be under-repair. Depending on the type of standby component the failure rate of the component can change when it is brought into operation. A common-cause event can cause more than one component failure in the system, meaning that component failure is not independent. For example, failure/repair processes can become dependent when there is a single maintenance engineer employed to handle the maintenance of many components of a system. This can result in a queue of components in need of repairs.

**Discrete Markov Chains**

**Transition Probability Matrix**

The Transition Probability Matrix (TPM) is used to determine the probability of being in a particular state after a certain number of time intervals. The matrix has elements $P_{ij}$,
which denotes the probability of making a transition to state \( j \) after a given time interval from state \( i \) at the beginning of the interval. The TPM for the two-state system shown in Figure 1.9 is given in equation 1.4.35. The size of the matrix is \( N \times N \), where \( N \) is the total number of states.

\[
[P] = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}
\]

(1.4.35)

When evaluating time dependent systems, the TPM is multiplied by itself \( n \) times, where \( n \) is the number of time intervals. The elements of \( [P]^n \) are \( P_{ij}^n \), the probability of being in state \( j \) after \( n \) time intervals given that it began in state \( i \). This form is given in equation 1.4.36.

\[
P(n) = P(0) \cdot [P]^n
\]

(1.4.36)

Where \( P(0) \) is the initial probability vector and is a row vector containing the probability of starting in each state.

When considering steady-state probabilities, \( P(\infty) \) these would not change with further multiplication, therefore equation 1.4.37 would be used for these types of systems.

\[
P(\infty)[P] = P(\infty)
\]

(1.4.37)

**Continuous Markov Processes**

When considering systems that are discrete in space and continuous in time, these can be modelled using continuous Markov processes. For the approach to be valid the system must be stationary, the process must lack memory and the states of the systems must be identifiable.

**State Equations**

There are two ways in which the state equations can be acquired; one way is by using the state diagrams. Taking the state diagram in Figure 1.9, the state equations are found by the following method:

\[
\frac{dP_{\text{state}}}{dt} = (\text{rate of entering state}) - (\text{rate of leaving state})
\]

(1.4.38)

Where \( P_{\text{STATE}} \) is the probability of being in a state.

From this statement equation 1.4.39 and 1.4.40 are formed.
\[
\frac{dP_w(t)}{dt} = -\lambda P_w(t) + \nu P_f(t) \quad (1.4.39)
\]

\[
\frac{dP_f(t)}{dt} = \lambda P_w(t) - \nu P_f(t) \quad (1.4.40)
\]

Where \(P_w(t)\) and \(P_f(t)\) are the probability that the component is working or failed at time \(t\), respectively.

In matrix form equations 1.4.39 and 1.4.40 become:

\[
\begin{bmatrix}
\dot{P}_w(t), \dot{P}_f(t)
\end{bmatrix} =
\begin{bmatrix}
P_w(t), P_f(t)
\end{bmatrix}
\begin{bmatrix}
-\lambda & \lambda \\
\nu & -\nu
\end{bmatrix}
\quad (1.4.41)
\]

Generalised, this equation becomes:

\[
[\dot{P}] = [P][A] \quad (1.4.42)
\]

Where \([A]\) is the transition rate matrix.

\[
[A] = \begin{cases}
a_{ij} & \text{Transition rate from } i \to j \\
a_{ii} - \sum_{j=1, j\neq i}^{p} a_{ij}
\end{cases}
\quad (1.4.43)
\]

Where \(p\) is the maximum number of states.

The transition rate matrix has certain properties that make it easy to develop:

- The number of states in the diagram equals the number of rows and columns in the matrix.
- The sum of each row of the matrix equals zero.
- Every non-diagonal elements in row \(i\) and column \(j\) represents the transition from state \(i\) to state \(j\).
- Diagonal elements \(i, i\) is the transition rate out of state \(i\). This is always negative.

The sum of any system state probabilities, at any time, must be equal to one, as shown in equation 1.4.44.

\[
\sum_{i=1}^{N_s} P_i(t) = 1 \quad (1.4.44)
\]

Where \(N_s\) is the total number of states.
Another method for finding the state equations is by denoting the state of the component at time \( t \) by:

\[
x(t) = \begin{cases} 
1 & \text{Failed} \\
0 & \text{Working} 
\end{cases}
\]  

(1.4.45)

Finding the probability that a component is in a failed state after \( dt \), requires knowledge of only the state of the component at the present time. Equation 1.4.46 defines that for \( P[x(t + dt) = 1] \) that, either the component was working at time \( t \) and failed in time \( dt \) or that the component had failed in time \( t \) and remained so during time \( dt \).

\[
P[x(t + dt) = 1] = P[x(t) = 0] \lambda dt + P[x(t) = 1](1 - \nu dt)
\]  

(1.4.46)

Equation 1.4.46 can be written as

\[
P_f(t + dt) = P_w(t)\lambda dt + P_f(t)(1 - \nu dt)
\]  

(1.4.47)

Rearranging the above gives equation 1.4.48.

\[
\frac{P_f(t + dt) - P_f(t)}{dt} = P_w(t)\lambda - P_f(t)\nu
\]  

(1.4.48)

As \( t \to 0 \), equation 1.4.48 becomes equation 1.4.40. Using the statement \( P_w(t) + P_f(t) = 1 \) and using the initial conditions \( P_f(0) = 0 \), equation 1.4.49 gives the unavailability.

\[
P_f(t) = \frac{\lambda}{\lambda + \nu}(1 - e^{-(\lambda+\nu)t})
\]  

(1.4.49)

Using the same process, but starting with \( P[x(t + dt) = 0] \) leads to equation 1.4.39 which can be solved to give the availability of a component.

\[
P_w(t) = \frac{\lambda}{\lambda + \nu} + \frac{\lambda e^{-(\lambda+\nu)t}}{\lambda + \nu}
\]  

(1.4.50)

In general equation 1.4.42 gives a set of \( N_S \) first order differential equations to solve. In some cases it may be possible to solve these using Laplace Transforms but in most situations numerical methods are adopted. Equation 1.4.42 can be expanded to give equation 1.4.51.

\[
\begin{bmatrix} \dot{P}_1 & \dot{P}_2 & \cdots & \dot{P}_{N_s} \end{bmatrix} = \begin{bmatrix} P_1 & P_2 & \cdots & P_{N_s} \end{bmatrix} [A]
\]  

(1.4.51)

Approximating

\[
\dot{P}_1(t) = \frac{P_1(t + dt) - P_1(t)}{dt}
\]  

(1.4.52)

Equation 1.4.51 can be written as equation 1.4.53.
\[
\begin{bmatrix}
P_1(t + dt) & P_2(t + dt) & \cdots & P_{N_s}(t + dt)
\end{bmatrix}
= \begin{bmatrix}
P_1(t) & P_2(t) & \cdots & P_{N_s}(t)
\end{bmatrix} \begin{bmatrix} I + [A]dt \end{bmatrix}
\] (1.4.53)

This can be written generally as equation 1.4.54.

\[
[P(t + dt)] = [P(t)][K]
\] (1.4.54)

Where \([K] = [I + [A]dt]\).

1.4.3 Simulation

1.4.3.1 Monte Carlo

Monte Carlo methods can be used when methods such as Markov and fault tree analysis are not applicable. For example, if the components or sub systems are dependent then fault trees cannot be used and if components do not have constant failure and repair rates, then Markov methods cannot be used. The use of Monte Carlo simulation requires no assumptions to be made regarding system behaviour. The following information is based on Andrews and Moss (2002) which also contains further reading on this topic.

Uniform Random Numbers

The Monte Carlo simulation method is dependent on the generation of random numbers which form a uniform distribution. A number of methods can be used to generate these numbers. One method by Von Neumann, the mid-square method, takes a random number, squares the value and then takes the middle numbers. These middle numbers form the new random number. For example squaring a starting number of 7989 gives 63,824,121. Taking the four middle numbers, the new value is 8241. A disadvantage of this method is that if zero is encountered then the sequence ends. Another disadvantage is that these values are not truly random, but pseudo-random; although they are uniform, the sequence is not random. Another method is random number tables. These tables are a sequence of random digits where entry can be from any point in the table and any subsequent values can be obtained by reading across or down the table.

Real engineering simulations require the generation of large quantities of random numbers and the only way to achieve this is to use a computer. Computers can generate pseudo-random number sequences.

The recursion formulae most commonly used are linear congruential generators. These have the form:

\[
x_{n+1} = (ax_n + b)(mod\ m)
\] (1.4.55)
Figure 1.10: Two-component system

Where $a$, $b$ and $m$ are positive integers.

Following an arbitrary number of iterations, $i$, of equation 1.4.55, the resulting random number is found as follows:

$$R_i = \frac{x_i}{m}$$  \hspace{1cm} (1.4.56)

Where $R_i$ is the random number produced in the range $[0,1]$, $x_i$ is the $ith$ number produced and $x_0$ is the seed.

The seed is a number that is specified initially to generate a sequence of pseudo-random numbers. As the same seed would produce the same sequence if repeated, $a$, $b$ and $m$ are chosen so that a large sequence of numbers are produced before the sequence produces the seed value again and it is repeated.

**Direct Simulation**

This method of modelling uses direct statistical simulation. For this type of system reliability simulation, two inputs are required: first is the statistical distributions for time to failure and time to repair for each component. The second input is the system logic, which includes how the components are connected and what the effects each component failure has on the system performance. The system is simulated by using random samples from the statistical distributions and tracking how the system functions when the component states change.

This method can be demonstrated by considering the two-component system given in Figure 1.10. Components A and B both have a probability of failure of 0.15. For there to be system failure both components A and B are required to fail.

The simulation is carried out by generating for each component in the system a random number that will be used to say if the component is working or failed. If the random number is less than the probability of failure, then the component is assumed to be failed. If the random number is larger than the probability of failure, then the component remains
working. This can be seen in Figure 1.11. The state of the system can then be determined.

**Distributions for generating event times**

**Exponential distribution**

The exponential distribution is the first of three distributions considered here. This distribution uses the density function presented in equation 1.4.57.

\[
f(t) = \frac{1}{\mu} e^{-t/\mu} \quad (1.4.57)
\]

Where \( \mu \) is the mean.

To obtain the random samples, a number of steps are followed: the first is to integrate the density function to obtain the cumulative failure distribution, \( F(t) \), as seen in equation 1.4.58.

\[
F(t) = \int_0^t f(u)du = 1 - e^{-t/\mu} \quad (1.4.58)
\]

As the cumulative failure distribution has the same range and properties as the random number distributions, the next step is to generate random numbers, \( X \), and equate to \( F(t) \) \((0 \leq F(t) \leq 1)\) giving equation 1.4.59.

\[
X = 1 - e^{-t/\mu} \quad (1.4.59)
\]
To find the distribution to represent time to failure, the equation is rearranged for $t$.

$$ t = -\mu \ln(1 - X) \quad (1.4.60) $$

If $X$ is assumed uniform over $[0, 1]$ then $1 - X$ must also be. This simplifies equation 1.4.60 to equation 1.4.61.

$$ t = -\mu \ln X \quad (1.4.61) $$

**Weibull distribution**

Like the exponential distribution, the Weibull distribution can also obtain the random samples directly. The density function that represents this distribution can be seen in equation 1.4.62.

$$ f(t) = \beta t^{\beta - 1} \eta^{-\beta} e^{-\left(t/\eta\right)^\beta} \quad (1.4.62) $$

Where $\eta$ is the Weibull scale parameter or characteristic life and $\beta$ is the Weibull shape parameter. Where $\beta < 1$ represents a reducing hazard rate, $\beta = 1$ is constant hazard rate and $\beta > 1$ is increasing hazard rate.

As with the exponential distribution, the density function is integrated to give the cumulative distribution, $F(t)$, as seen in equation 1.4.63.

$$ F(t) = 1 - e^{-\left(t/\eta\right)^\beta} \quad (1.4.63) $$

Once again the cumulative distribution has the same range and properties as the random number distribution and so from the generated random numbers, $X$, equation 1.4.64 is used.

$$ X = 1 - e^{-\left(t/\eta\right)^\beta} \quad (1.4.64) $$

By rearranging the equation above for time, $t$, to represent the time to failure (or repair), equation 1.4.65 is obtained.

$$ t = \eta[-\ln(X)]^{1/\beta} \quad (1.4.65) $$

**Normal distribution**

The normal distribution is the only distribution considered here for which the density function cannot be integrated to obtain the cumulative distribution. This is due to the density function as seen in equation 1.4.66.
Chapter 1. Introduction

\[ f(t) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2}[\frac{t-\mu}{\sigma}]^2} \] (1.4.66)

Where \( \sigma \) is the standard deviation and \( \mu \) is the mean.

Instead, a method using the central limit theorem is introduced. The central limit theorem states that if \( X_1, X_2, \ldots, X_n \) are \( n \) independent random variables which are identically distributed and have mean, \( \mu \) and variance, \( \sigma^2 \) and then \( S_n \) is the sum of all these variables then the random variable \( (S_n - n\mu)/(\sigma\sqrt{n}) \) is asymptotically normally distributed with a mean of 0 and a standard deviation of 1. \( S_n \) can use random numbers \( U(0,1) \) to represent it as these are identically distributed, however it should be noted that \( S_n \) will only ever be approximately normal for a finite value of \( n \). Therefore to find this value of \( n \), values are tested to see what kind of distribution is obtained. Starting with \( n = 2 \), the distribution is a triangular distribution which is unsuitable. For \( n = 3 \), a bell-shaped distribution is obtained. This is closer to what is required, therefore \( n \geq 3 \) is suitable. A number that is mathematically suitable is \( n = 12 \), as \( X_i \) has \( \mu = 0.5 \) and \( \sigma^2 = 1/12 \), therefore \( S_n \) is \( N(6,1) \). Therefore to obtain a random sample from the normal distribution, twelve random numbers are required. \( X \) is the summation of these numbers, as given in equation 1.4.67.

\[ X = \sum_{i=1}^{12} X_i \] (1.4.67)

The central limit theorem states \( X \) is normally distributed with mean 6 and standard deviation 1. Hence \( t \), given by applying this equation 1.4.68, is normally distributed with mean \( \mu \) and standard deviation \( \sigma \).

\[ t = (X - 6)\sigma + \mu \] (1.4.68)

1.4.3.2 Petri Nets

Carl Adam Petri, in his 1962 Ph.D. Thesis entitled *Kommunikation mit Automaten* (Communication with automata), presented a new graphical representation of systems. The idea presented in the thesis was to show the theory of communication between non-simultaneous components of a computer system. Petri focused on the description of the relationships between events. Anatol Holt et al (1968) produced the translation to Petri’s dissertation and added to the work considerably; this was given in the final report of the Information System Theory Project. Holt and Commoner (1970) wrote a paper entitled *Events and Conditions*. This paper was concerned with the part-by-part performance of a system, and was chiefly concerned by concurrent operations. This paper was also an important part of the early research of Petri nets.
Petri nets are useful when simulating large systems when the computational time to analyse a system becomes too significant to ignore. They are incredibly versatile in terms of the system features that they are capable of modelling. This section discusses the basic definitions of Petri nets, and with given examples, how Petri nets are applicable to this work.

There are four basic components that constitute a Petri net; places, transitions, arcs (also referred to as edges) and tokens. A graphical representation of each is given in Table 1.3. Places can represent states of a system, such as the Petri net in Figure 1.12 that shows a system in one of two states; working ($P_1$) or failed ($P_2$). This is one of the simplest examples of a Petri net, but these can be expanded considerably to include other system states such as under repair, system down, repaired and system standby. Places are not limited to just representing states; they can also represent components within a system, or a member of the workforce carrying out maintenance. This makes Petri nets very versatile and so can accommodate the needs of many different problems.

Table 1.3: Key to Petri nets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Place" /></td>
<td>Place</td>
</tr>
<tr>
<td><img src="image" alt="Transition" /></td>
<td>Transitions (delayed and instant)</td>
</tr>
<tr>
<td><img src="image" alt="Arc" /></td>
<td>Arc</td>
</tr>
<tr>
<td><img src="image" alt="Token" /></td>
<td>Token</td>
</tr>
</tbody>
</table>

![Figure 1.12: Working and failed state system](image)

Tokens, represented as a small dot, are located inside places and used as locators. For
example, in Figure 1.12 a token can be seen inside place, $P_1$. This indicates that the system is in a working state. If there were a token in place $P_2$, this would indicate that the system is in a failed state. As tokens can be treated as a finite resource, they can also be used to signify the number of components available or the number of maintenance engineers available to repair a system. It should also be noted that places can hold more than one token.

Arcs, sometimes referred to as edges, are used to show the link between places and transitions. If there is more than one arc connecting a place to a transition then this is represented on a Petri net as a slash through the arc with a number by it indicating the number of arcs. An example of this can be seen in Figure 1.13.

Transitions dictate at which “time” a system “moves” from one state to another. In Figure 1.12, the transition $t_F$ would represent the ‘time to failure’ of the system. The rules that govern when a transition can fire and tokens are moved, are described in detail later.

**Formal definition of a Petri net**

The formal definition for a Petri net is given in Schneeweiss (1999):

$$G_{PN} = (V_p, V_t, E; M(0), D, W)$$

Where $G_{PN}$ is the Petri net graph, $V_p$ is the set of places, $V_t$ is the set of transitions, $M(0)$ is the initial marking vector, $D$ is the vector of switching delays (transition times) and $W$ is the vector of weights of edges i.e. the number of arcs making a given connection, $E$ is the set of edges (ordered pairs of nodes), where $E \subseteq (V_p \times V_t) \times (V_t \times V_p)$. The initial marking vector, $M(0)$, lists the number of tokens in each place when the system is initialised.

For example, for the Petri net shown in Figure 1.14:

$$G_{PN} = (V_p, V_t, E; M(0), D, W)$$

$V_p = \{1, 2, 3, 4, 5, 6\}$
$V_t = \{1, 2, 3, 4, 5, 6\}$
$E = \{(p_1, t_1), (p_1, t_2), (t_1, p_3), (t_2, p_2), (p_2, t_3), (t_3, p_4), (p_4, t_4), (t_4, p_3), (p_5, t_5), \}$
1.4. Reliability Techniques

\[(t_5, p_4), (p_3, t_6), (p_4, t_6), (t_6, p_6)\}\n
\[M(0) = (1, 0, 0, 0, 1, 0)\]
\[D = (D_1, D_2, D_3, D_4, D_5, D_6)\]
\[W = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)\]

Figure 1.14: Petri net

The Rules of Execution

The movement of tokens between places within a Petri net represents its dynamic behaviour. In reliability modelling the position of tokens in the Petri net at a given instant in time, known as its marking, represents a particular system state.

A set of rules for the movement of tokens are given as follows:

- A Petri net executes by firing transitions.
- A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places.
- A transition may fire if it is enabled.
  - A transition is enabled if each of its input places has at least as many tokens in it as arcs from the place to the transition.
  - Transitions can have a time delay associated within them; such transitions are known as timed transitions. The time for the delay only passes while tokens are present, as described above.
- A transition fires by removing enabling tokens from its input places. One token is removed from each input place for each arc connecting the place to the transition.
After a transition has fired, tokens are deposited into all places connected as outputs from the transition. One token is deposited for each arc connecting the transition and the output place. Note that output arcs can be connected to input places, meaning that tokens are deposited back into the input place.

In the example Petri net shown in Figure 1.15a, \( t_i \) terms denote the times at which the transitions fire. The transition sequence can be expressed in the following form, showing the order in which these transitions fire: \( t_1 \to t_2 \to t_5 \to t_4 \to t_3 \to t_6 \). The movement of the tokens in the system as each transition fires can be seen in Figure 1.15. Another way in which this sequence can be written would be as follows: \( t_1 < t_2 < t_5 < t_4 < t_3 < t_6 \). This expresses, using an inequality, the relative time values used in the transitions.

**Fault Trees to Petri Nets**

Fault tree gates can be represented by Petri nets such as those shown in Table 1.4 (For a full list see Liu and Chiou (1997)), hence enabling fault trees to be converted into Petri nets.
1.4. Reliability Techniques

Table 1.4: Fault tree symbols and Petri net equivalent

<table>
<thead>
<tr>
<th>Gate name</th>
<th>Fault tree symbol</th>
<th>Petri net equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND gate</td>
<td>![AND gate diagram]</td>
<td>![AND gate Petri net]</td>
</tr>
<tr>
<td>OR gate</td>
<td>![OR gate diagram]</td>
<td>![OR gate Petri net]</td>
</tr>
</tbody>
</table>

Other aspects of the system can be incorporated into Petri nets, such as maintenance policies for each component. The dependencies within the system can also be shown by changing a fault tree to a Petri net.

Table 1.4 gives an example of two of the main gates that are commonly converted from fault trees to Petri nets, the AND gate and the OR gate. The AND gate requires that both $P_1$ and $P_2$ have a token to enable the immediate transition labelled $T_1$ so it can fire. The OR gate requires either place $P_1$ or $P_2$ to have a token to enable transition $T_1$.

**Minimal Cut Sets and Path Sets**

Minimal cut sets and path sets are usually associated with fault trees, but can be applied in the same way with Petri nets. Path sets are the opposite of cut sets in that they show the minimal, but sufficient components in order for the system to be successful. There are multiple methods by which this can be achieved. Liu and Chiou (1997) show that minimal cut sets and path sets can be determined by the following procedure:

1. If a place representing a basic event is connected via arcs and transitions to the place representing the top event, where the only input place to the transitions is this basic event, then this basic event is a minimal cut set.

2. If a path between a basic event and the top event places includes transitions that involve further places, then identify these further places.

3. If the places identified in the previous step do not represent basic events, then apply steps 1 and 2 again to each place to identify the basic events.
4. Further basic events identified in step 3 form cut sets. Transitions which involve more than one basic event give rise to cuts sets of multiple basic events. Once a full list of cut sets is obtained these can be reduced to minimal cut sets by removing any cut sets which comprise solely of smaller cut sets.

The procedure for obtaining the minimal cut sets and path sets for a fault tree have a total of eight steps, as opposed to the four given above. This makes Petri nets much more efficient.

**Dual Petri Nets**

To find the path sets of a Petri net, the above procedure would be used, but on a dual Petri net. The dual Petri net switches the transitions with multiple inputs to a single input and vice versa. The equivalent fault tree is changing AND to OR gates and vice versa. As an example Figure 1.16b shows a representation of the dual Petri net presented in Figure 1.16a.

**Matrix Method**

Another way to obtain minimal cut sets and path sets is by use of a matrix method. This matrix method begins with the place that represents the top event and moves down to the basic events. Liu and Chiou (1997) describe the procedure:
1.4. Reliability Techniques

1. If there are multiple arcs connected to an output place from multiple transitions then write the input place identifiers in a horizontal list.

2. If a single arc is connected to an output place then write the input place identifiers in a vertical list.

3. Once all lists are represented as basic events then the matrix is established. If there is common basic event located between rows or columns, it is the basic event shared for each row or column. The column vectors of the matrix represent cut sets while row vectors path sets.

4. Taking the full list of cut sets and path sets these can be reduced to minimal cut sets and path sets by removing any cut sets or path sets which comprise solely of smaller cut sets or path sets.

This is an efficient method of identifying the minimal cut sets and the path sets without needing to change the Petri net to the dual Petri net.

Absorption

Absorption can occur in Petri nets when the firing time is not required, i.e. when there is no time between input and output. Absorption removes any non-required intermediate steps. Examples of this can be seen in Figure 1.17a and Figure 1.17b. Figure 1.17a shows that the input can go directly to the output. As soon as the transitions are enabled they fire the tokens from the current place to the recipient of the transition. The figure shows that there is a chain of one place followed by one transition followed by a place and so on. As there is no delay time associated with the transitions the token would move instantaneously from the first place to the last. As there are no other places connected to the transitions, it means that there is only one outcome from such a situation: the token would always go from $P_1$ to $P_3$, therefore it can be shown as a single place.

Figure 1.17b shows a Petri net with hierarchical transitions which consists of multiple inputs can be combined to one transition. This Petri net shows that the places $P_1$ and $P_2$ require a token in order to enable the transition that would place a token in $P_4$. Then, should there be a token in $P_3$, transition 2 would fire to deposit a token into $P_5$. To obtain a token in $P_5$, the minimal number of tokens in other places that have no transitions to them is 3. One token would be required in $P_1$, $P_2$ and $P_3$; $P_4$ is an intermediate place and would not require a token. Therefore $P_4$ can be absorbed.

Marking Transformation

Another way to represent a Petri net is by using marking, as explained in Liu and Chiou (1997). Marking is a way to represent a Petri net as a vector, $M$, representing the number
of tokens in each place. The vector therefore describes completely the state of the Petri net at a given time. When transitions in the Petri net fire, this vector is transformed using the incidence matrix, $A^T$. Rows of $A^T$ are associated with places in the Petri net (in the same order as they appear in $M$), while columns represent the transitions. Each column of $A^T$ shows the net change to the number of tokens in each place when a given transition fires. Equation 1.4.70 shows the relationship between the incidence matrix and the next state, given that it is in the $k$th state. $S$ represents the firing times, $T_i$, of the $i$th transition.

$$M_{k+1} = M_k + A^T S$$ \hspace{1cm} (1.4.70)

To combine all the marking transformations to the final marking $M_n$ to the initial marking, $M(0)$, equation 1.4.70 is rewritten to give equation 1.4.71.

$$M_n = M_0 + A^T \sum$$ \hspace{1cm} (1.4.71)

Rearranging the above gives:

$$A^T \sum = \Delta M = M_n - M_0$$ \hspace{1cm} (1.4.72)

Where $\sum$ denotes the firing counter. This is a vector with an element for each transition in the Petri net; transitions which are enabled are denoted by 1. This vector, therefore, selects which transitions from matrix $A^T$ are fired during the transformation of matrix $M$.

To demonstrate this, the Petri net in Figure 1.16a was used. The incidence matrix for this Petri net is given in equation 1.4.73. With an initial marking of equation 1.4.74, and a
1.4. Reliability Techniques

transition sequence of \( t_1, t_3, t_4 \), i.e. the firing counter given in equation 1.4.75, \( M_n \) would be as that given in equation 1.4.76, through application of equation 1.4.71. Different firing counters will produce different final markings, \( M_n \).

\[
\begin{align*}
T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
P_1 & \begin{bmatrix} -1 & 0 & 0 & 0 \end{bmatrix} \\
P_2 & \begin{bmatrix} 0 & -1 & 0 & 0 \end{bmatrix} \\
P_3 & \begin{bmatrix} 0 & 0 & -1 & 0 \end{bmatrix} \\
A^T = P_4 & \begin{bmatrix} 1 & 1 & -1 & 0 \end{bmatrix} \\
P_5 & \begin{bmatrix} 0 & 0 & 1 & -1 \end{bmatrix} \\
P_6 & \begin{bmatrix} 0 & 0 & 0 & -1 \end{bmatrix} \\
P_7 & \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]

(1.4.73)

\[
M(0) = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}^T
\]

(1.4.74)

\[
\sum_1 = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}^T
\]

(1.4.75)

\[
M_n = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T
\]

(1.4.76)

Reachability Graphs

Reachability graphs are used when multiple transitions in a Petri net can fire simultaneously. In such situations it is not possible to determine which transition fires first; if two transitions rely on the same input tokens then only one can fire. Reachability graphs show the current state of the Petri net and branch out to different possible states when such situations arise. In Figure 1.18a an example of a simple Petri net is given, and in Figure 1.18b is the equivalent reachability graph.

Figure 1.17a shows a simple Petri net with transition \( T_1 \) enabled. As seen in Figure 1.17b, the starting point at the top of the graph is the initial markings of the Petri net, and below this is an arrow to represent the transition firing, leading to the next marking data. This marking data shows that there are now three transitions enabled of which only one can fire. This means that three branches are produced to signify the outcomes of each of the three transitions firing. This process is continued down each of the branches until there is no more transitions left enabled. The final reachability graph is given in Figure 1.17b.

Although for this example the process reached a finite set of markings, sometimes this process can be infinite. This can be reduced to a more manageable size by using the symbol \( \omega \), which can be taken as the symbol for \( \text{infinity} \). This can also be used for
any large number of tokens. This symbol is located in the marking(s), which create this infinite process. More information on this and an algorithm that can automate this process of creating a reachability graph is given by Peterson (1981).

1.5 Summary

This Chapter has introduced the possible reliability methods available that could be used in the process considered here. From the research, only a few of the methods available are suitable for a variety of systems. Although Fault Tree Analysis is the most widely used method, this would be inappropriate for the work presented here as fault trees cannot handle dependencies. From the research, Markov methods and Petri nets would be the most suitable. However as Markov models, more so than Petri nets, are susceptible to state explosion, this would make the work more inefficient. Petri nets are very powerful and flexible and seem well suited to the application discussed in this thesis. However, before a method can be chosen the methods presented in this chapter need to be considered for their usability within the scope of Phased Mission Systems, which is discussed in the next chapter.
2.1 Introduction

There are many applications where missions are required, such as in the Aerospace and Nuclear Industries. Missions consist of phases which can be considered as different tasks that a system must undertake and accomplish for the mission to be successful. Within these phases the system can have a different configuration as the success criteria for each phase can be different. The components within that system can also have different failure behaviours from phase-to-phase (Xing & Dugan 2002).

A good example of a phased-mission is an aircraft flight. The mission would be to take passengers from one airport to another. This mission could consist of seven phases; taxiing to the runway, take-off, climb, descent, landing and taxiing to the terminal. A graphical representation of such a flight is given in Figure 2.1. Each phase of the mission requires a different configuration of the aircraft and therefore the probability of system failure must be calculated for each phase in order to obtain the mission failure probability. A model therefore is required for each phase.
2.1.1 Types of phased-mission systems

2.1.1.1 Static and Dynamic Phased-Mission Systems

A system is said to be static if the failure of the mission, in any phase, is dependent on combinations of the component failure events, i.e. if the structure of the reliability model in any phase is combinatorial. A system is said to be dynamic if the failure of the mission in any phase is dependent on the combinations of component failure events and the order in which input events occur, i.e. if the order in which component failure events occurs affects the outcome (Xing & Amari 2008).

2.1.1.2 Repairable and non-repairable Phased-Mission Systems

In non-repairable Phased-Mission Systems (PMSs), once a component has failed in a phase, it has failed for every other succeeding phase. For a repairable PMS there are two things to consider; the failure characteristics of a component and the maintenance plan in place. Meshkat (2000) investigated the following:

- **Time-driven maintenance** (Scheduled maintenance): Maintenance is initiated on a predetermined schedule.
- **Failure-driven maintenance** (Unscheduled maintenance): Maintenance is initiated when a component failure occurs.
- **Condition-driven failure**: Maintenance is initiated when a component fails, but the system does not. However if the system fails there can be no repair on the system; this is the difference between failure- and condition-driven maintenance.

2.1.1.3 Coherent and non-coherent Phased-Mission System

A PMS is coherent when each component of a system contributes to the state of said system. For every component that fails the system state cannot improve, it can only
worsen or remain the same (Andrews & Beeson 2003). A non-coherent PMS, however, can worsen or improve with the functioning or failure of a component, respectively. Non-coherent PMSs can be represented using non-coherent fault trees which are distinguished by the use of inverse gates (a NOT gate) as well as the usual logic gates (e.g. AND gate).

2.1.2 Analytical Modelling Techniques

There are multiple ways in which a phased-mission system can be evaluated. There are two areas into which these fall; analytical modelling and simulation. Simulation is often found to cost more in terms of computational requirement, but does give a better generality in system representation (Smotherman & Zemoudeh 1989). Analytical modelling provides a direct solution, however it can be very difficult to generate analytical models of complex systems. Analytical models can be broken down into three groups (Xing & Amari 2008):

- **State-space orientated models**: Examples of state-space approaches are Markov chains and Petri nets. Each is flexible and can model complex dependencies in system components. State space models can be used for both dynamic and static phases.

- **Combinatorial models**: These methods assume that all the components in a system, in each phase, fail statistically(s)-independently, so should they fail it would have no bearing on whether another component in the system fails or not. This s-independence is dealt with across the phases for a given component. Examples of combinatorial approaches are fault trees (mini-component systems and Boolean algebraic method) and BDDs. Combinatorial models can only be used when phases are static, as discussed above in section 2.1.1.1.

- **Phase modular solution**: This takes advantage of both of the above methods by addressing their limitations. Combinatorial models are computationally efficient, but can only be used when the phases are static. State-space models such as Markov chains have to be used if any phase in a mission is dynamic. A problem with Markov chains is that state explosion can occur, making the Markov approach computationally intensive. Therefore the phase modular solution uses both BDD and Markov chain solution when appropriate.

2.2 Non-Repairable Systems

2.2.1 Phase Fault Trees

2.2.1.1 Quantitative Analysis

A technique for an exact unreliability solution for a phased-mission system is given by Esary and Ziehms (1975). The method is based around components that, once failed,
cannot be repaired or replaced. Also the system can only be either functioning or failed. The method given reduces a multi-phased system into a single equivalent phase system. Each phase of a mission can be expressed as either a RBD or a fault tree, as it is assumed that each phase configuration is coherent. Presented in Burdick et al. (1977) is Esary and Ziehms technique given in five steps:

1. **Mission cut-set cancellation**: Any minimal cut set of a phase is removed if it contains a minimal cut set for a later phase. An example of this is if there is a two phase system as shown in Figure 2.2a, where the dotted OR gate represents the input into a phased mission top event. The minimal cut sets for the fault tree are as follows:

   - Phase one: \{A.C\} and \{B.C\}
   - Phase two: \{D\}, \{B.C\} and \{E\}

   From these there is a common minimal cut set between the two phases, \{B.C\}. From the rule above \{B.C\} is removed from phase one as it occurs in the second phase. Therefore phase two remains the same, however phase one can now be represented as the fault tree in Figure 2.2b.

2. **Basic-event transformation**: For a \(j\)-phase mission, the series logic with basic events \(C_{k1}, \ldots, C_{kj}\), which perform statistically independently with the failure probability replace basic event \(C_k\). Where \(C_{k1}\) is the basic event \(k\) occurring in phase 1. Taking the example from above, each side of the fault tree in Figure 2.2b can now be represented as the fault tree in Figure 2.2c. In the first phase, a subscript of one denotes that the basic event in that phase. In phase two an extra OR gate exists for each basic event so that one branch represents the basic event in phase one and another represents the basic event in phase two.

3. The configuration of the phases can now be considered as a new system with sub-systems operating in series logic and can also be considered, as a single phase mission. An example of this can be seen in Figure 2.2d.

4. As with any other fault tree, the minimal cut sets can now be found for the new logic model. For the example considered, all cut sets are minimal.

5. Normal quantitative analysis techniques are used to find the unreliability for the system.

Esary and Ziehms (1975) also presented new unreliability equations for phased mission systems (reviewed in Burdick et al. (1977)). These particular approximations are designed for non-repairable systems and can be used for larger systems as they reduce the cost of calculating the exact unreliability.
2.2. Non-Repairable Systems

Figure 2.2: Process for a two phase mission to find the exact solution
1. **IN-EX method**: The IN-EX method first identifies the minimal cut sets for each phase of a mission. These are then used to calculate the unreliability of phase $i$, $Q_i$, by using the inclusion-exclusion expansion given earlier in equation 1.4.3. Then either equation 2.2.1 or equation 2.2.2 can be used to calculate the reliability of a mission with $m$ phases or the unreliability of the mission, respectively.

$$R_{IN-EX} = \prod_{i=1}^{m} R_i \quad (2.2.1)$$

$$Q_{IN-EX} \leq \sum_{i=1}^{m} Q_i \quad (2.2.2)$$

2. **IN-EX-CC method**: This is similar to the IN-EX method, the difference is with the mission cut set cancellation. The cancellation, in step 1 from above, is done before $Q_j$ is calculated for each phase. For this bound the value of $Q_j$ will be generally less than the $Q_j$ found in the IN-EX method (fewer cut sets).

3. **MCB method**: This method requires four steps; the first is to obtain the minimal cut sets for each phase, using the appropriate model. The second step is to calculate $q_{ij}$ for a minimal cut set $i$ in phase $j$ using equation 2.2.3

$$q_{ij} = \prod_{l=1}^{k_{ij}} Pr\{C_l\} \quad (2.2.3)$$

Where $C_l$, $l = 1, \cdots, k_{ij}$ are basic events in cut set $i$ in phase $j$.

The third step is to estimate $R_j$ using the minimal cut bound as given in equation 2.2.4.

$$R_j = \prod_{i=1}^{n_j} r_{ij} \quad (2.2.4)$$

The fourth and final step is to calculate $R_{MCB}$, which is the same as that given in equation 2.2.1.

4. **MCB-CC method**: This method has the same steps as that given for the MCB method, but with an additional step between the first and the second step that carries out the mission cut-set cancellation as discussed earlier.

The relationship between these methods can be seen in equation 2.2.5.
2.2. Non-Repairable Systems

\[
Q \leq Q_{IN-EX-CC} \leq \left\{ \begin{array}{l}
Q_{MCB-CC} \\
Q_{IN-EX}
\end{array} \right\} \leq Q_{MCB} \quad (2.2.5)
\]

Dazhi and Xiaozhong (1989) discuss a new set of Boolean algebra specifically for phased mission systems, taking into account initial conditions at the start of the phase. This method uses generalised intersection and union concepts in order to do this. Assuming that the system contains non-repairable events, that the model is coherent, the basic events are statistically independent in failure and that the transition time between two phases is instantaneous, equation 2.2.6. If \(1 \leq k \leq j\), where \(j\) and \(k\) are phase numbers:

\[
A_{(j)} = A_1 \cup A_2 \cup \cdots \cup A_j \\
= \bigcup_{i=1}^{k} A_i \cup \bigcup_{i=k+1}^{j} A_i \\
= A_{(k)} \cup \bigcup_{i=k+1}^{j} A_i \quad (2.2.6)
\]

Where \(A_{(j)}\) denotes that the basic event \(A\) exists in phase \(j\), given that it occurred between the first and \(j\) phase inclusive.

\[
A_{(k)} \cap A_{(j)} = A_{(k)} \cap \left( A_{(k)} \cup \bigcup_{i=k+1}^{j} A_i \right) \\
= A_{(k)} \cup \bigcup_{i=k+1}^{j} (A_{(k)}A_i) \\
= A_{(k)} \quad (2.2.7)
\]

\[
A_{(k)} \cup A_{(j)} = A_{(k)} \cup \left( A_{(k)} \cup \bigcup_{i=k+1}^{j} A_i \right) \\
= A_{(k)} \cup \bigcup_{i=k+1}^{j} A_i \\
= A_{(j)} \quad (2.2.8)
\]

To calculate mission unreliability the following is true: it can be said that if there is system failure in phase \(n\), then the system has failed in phase \(n\) or in any of the previous \(n - 1\) phases. This can be demonstrated mathematically using equation 2.2.9.
\[ X_{(n)} = X_1 \cup X_2 \cup X_3 \cup \cdots \cup X_n \] (2.2.9)

Where \( X_{(n)} \) represents system failure in phase \( n \), \( X_i \) represents the first failure in phase \( i \).

From this the mission unreliability is found using equation 2.2.10.

\[
Q_{miss} = P\left( X_{(n)} \right) \\
= P \left( \bigcup_{i=1}^{n} X_i \right) \\
= P \left[ \bigcup_{i=1}^{n} \left( \bigcup_{j=1}^{m_i} C_{(i)j} \right) \right] 
\] (2.2.10)

Where,

\[ X_i = \bigcup_{j=1}^{m_i} C_{(i)j} \] (2.2.11)

Where, \( C_{(i)j} \) represents a minimal cut set for \( X_i \), \( m_i \) represents the number of minimal cut sets in phase \( i \), \( n \) represents the number of phases.

La Band (2005) describes an analytical technique for the efficient representation and solution of phased-mission systems. Methods for representing the phase unreliability and the mission unreliability were considered for non-repairable systems.

The method introduced was to use fault trees to represent a failure event and in the case of phased-mission systems, multiple fault trees were drawn to represent each phase failure in a single mission. These fault trees show what components, or combination of components could cause that phase failure, as this can be different for each phase of the mission. La Band takes into account that a component can fail in any phase and although it may not cause a phase failure in that phase, it could cause a later phase to fail. A new way of using fault trees to show this was developed.

Each component in the fault tree is shown with a subscript of the phase that it resides in, e.g. in the second phase a subscript of 1 or 2 would be seen to signify that the component has failed in either phase 1 or phase 2; these would be connected through an OR gate. As this is a non-repairable system the OR gate signifies that once that component has failed in one phase it is failed for the rest of the mission.

To show the mission unavailability, the above representation was used for each of the \( i \) phases of the mission. For every phase, any and all previous phases are taken into account as the mission would fail if any of the phases failed. To take into account the success of the previous phases, the phase failure fault trees are used with a NOT gate to show that the
2.2. Non-Repairable Systems

previous phases did not fail. Each of these successful phases and the current phase failure are all connected to an AND gate. A generalised phase failure fault tree is given in Figure 2.3.

2.2.1.2 Qualitative Analysis

To find the failure modes of a system, defined earlier, the same process as for single phase fault trees is used. The NOT logic gate represents the non-coherence of the tree due to the requirement of noting the success of the previous phases. An example of a three-phase mission is given in Figure 2.4.

The phase failure fault tree for phase 1 remains unchanged for the analysis, as there are no previous phases to account success for. The phase 2 failure fault tree incorporates the success of the previous phase, and so the phase fault tree is represented by Figure 2.5. Finally phase 3 would incorporate the previous two phase successes to find the failure
probability in phase 3. This is represented by Figure 2.6.

![Fault Tree Diagram](image)

Figure 2.5: Phase fault tree construction of phase 2

Modularisation a useful technique to simplify the phase failure fault trees to decrease computational time. The modularisation technique by Reay and Andrews (2002), referred to as FAUNET reduction, is employed here. There are three mechanisms of modularisation and these are described as follows:

- **Contraction**: Gates of the same type connected in subsequent fashion are contracted to produce a single gate, so there is an alternating sequence of OR and AND gates within the tree structure.

- **Factorisation**: Events that occur in pairs under the same gate type are recognised and combined to form a single complex event. These complex events are given a numerical label starting from 2000 with increments of 1. The NOT logic gate requires De Morgan’s laws to be followed in that basic events occurring together in one gate type, e.g. AND (OR), must have complements that occur in the opposite gate type, e.g. OR (AND).

- **Extraction**: When there is a common basic event under a structure, like those shown in Figure 2.7, this is isolated and the other basic events brought together under the same gate.
2.2. Non-Repairable Systems

These steps are designed to be repeated in order until no further changes to the structure of the fault tree can be made. Such modules can be taken from the structure of the tree to create a sub-tree. These become completely independent to the rest of the tree and analysed separately to put the results into the higher-level fault tree (Reay & Andrews 2002).

For the example considered in Figure 2.4 the extraction method cannot be used. The reason is that there are no common basic events in the same branch of the tree that considers the failure of the individual phases. In Figure 2.6 there are no common basic events in the each of the three phases’ failure branches. Although basic event $D$ exists in both phase 2 and phase 3 failure branches, these cannot be merged as they are individually linked to each phase. In phase 2 failure, only phase 1 and phase 2 failure of basic event $D$ is
considered, whereas in phase 3 it is considered in all three phases. Also all previous phases are considered in a working condition, and the current phase in a failed state. Functioning and failed basic events cannot be merged.

### 2.2.1.3 Prime Implicants

Prime implicants are used to show the combination of basic events that lead to phase or mission failure. The notation used for failure and success of component, $A$, in phase $i$ is given as:

1. $A_i$ – Failure of component $A$ in phase $i$
2. $\bar{A}_i$ – Success of component $A$ in phase $i$

To show the failure, or success, of a component between the beginning of phase $i$ and the end of phase $j$, component $A$ would be expressed as:

1. $A_{ij}$ – Failure of component $A$ between phase $i$ and phase $j$
2. $\bar{A}_{ij}$ – Success of component $A$ from phase $i$ through to the end of phase $j$

New algebra laws developed by La Band (2005) are shown in Table 2.1.

Using the three phase mission example given earlier in Figure 2.4, and the subsequent individual phase failure fault trees in Figure 2.5 and Figure 2.6, the prime implicants for the three phases can be found as follows, where $T_i$ is the top event for phase $i$:

**Phase 1 Failure:**

$$T_1 = A_1 \cdot B_1$$ \hspace{1cm} (2.2.12)

Prime implicants = \{ $A_1 \cdot B_1$ \}

**Phase 2 Failure:**

$$T_2 = (\bar{A}_1 + \bar{B}_1) \cdot ((C_1 + C_2) \cdot (D_1 + D_2)) = \bar{A}_1 \cdot C_{12} \cdot D_{12} + \bar{B}_1 \cdot C_{12} \cdot D_{12}$$ \hspace{1cm} (2.2.13)

Prime implicants = \{ $\bar{A}_1 \cdot C_{12} \cdot D_{12}$, $\bar{B}_1 \cdot C_{12} \cdot D_{12}$ \}

**Phase 3 Failure:**
Table 2.1: Algebraic law for phased-mission systems for $i < j$

<table>
<thead>
<tr>
<th>Law Number</th>
<th>Algebraic Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A_i \cdot A_i = A_i$</td>
<td>Component $A$ fails in phase $i$ AND phase $i$. This is a repeated event.</td>
</tr>
<tr>
<td>2</td>
<td>$A_i \cdot A_j = 0$</td>
<td>Component $A$ fails in phase $i$ AND phase $j$. As these are mutually exclusive events they cannot both occur.</td>
</tr>
<tr>
<td>3</td>
<td>$A_i \cdot A_{ij} = A_i$</td>
<td>Component $A$ fails in phase $i$ AND between phase $i$ and phase $j$. As mutually exclusive events cannot occur together, i.e. component $A$ failure in phase $i$ AND any other phase between phase $i + 1$ and phase $j$, the common event, failure of component $A$ in phase $i$, is the result.</td>
</tr>
<tr>
<td>4</td>
<td>$\bar{A}_i \cdot A_i = 0$</td>
<td>Component $A$ is functioning in phase $i$ AND fails in phase $i$. The event and its complement cannot occur at the same time.</td>
</tr>
<tr>
<td>5</td>
<td>$\bar{A}_i \cdot A_j = A_j$</td>
<td>Component $A$ is functioning in phase $i$ AND fails in phase $j$. Failure in phase $j$ assumes that the component is successful in any previous phase, in this case phase $i$. Therefore the statement of component $A$ is working in phase $i$ is not required.</td>
</tr>
<tr>
<td>6</td>
<td>$\bar{A}<em>i \cdot A</em>{ij} = A_{i+1,j}$</td>
<td>Component $A$ works in phase $i$ AND fails between phase $i$ and phase $j$. The success and the failure of $A$ in phase $i$ cannot be combined as given by law 4, therefore the combination is the failure of component $A$ between phase $i + 1$ and phase $j$.</td>
</tr>
<tr>
<td>7</td>
<td>$\bar{A}<em>i \cdot \bar{A}</em>{i+1} \cdots \bar{A}<em>j = \bar{A}</em>{ij}$</td>
<td>Component $A$ works from phase $i$ up to and inclusive of phase $j$.</td>
</tr>
<tr>
<td>8</td>
<td>$A_i \cdot A_{i+1} \cdots A_j = A_{ij}$</td>
<td>Component $A$ has failed from phase $i$ up to and inclusive of phase $j$.</td>
</tr>
</tbody>
</table>
\[ T_3 = (\bar{A}_1 + B_1) \cdot (\bar{C}_1 \cdot \bar{D}_1 \cdot \bar{D}_2) \cdot (B_1 + B_2 + B_3) \cdot (D_1 + D_2 + D_3) \]
\[ = \bar{A}_1 \cdot \bar{C}_{12} \cdot B_1 \cdot D_{13} + \bar{A}_1 \cdot \bar{C}_{12} \cdot B_{23} \cdot D_{12} + B_2 \cdot D_3 + B_3 \cdot D_3 \] (2.2.14)

Prime implicants = \{\bar{A}_1 \cdot \bar{C}_{12} \cdot B_1 \cdot D_{13}\}, \{\bar{A}_1 \cdot \bar{C}_{12} \cdot B_{23} \cdot D_{12}\}, \{B_2 \cdot D_3\}, \{B_3 \cdot D_3\}

With each phase failure prime implicants established, the probability of phase and mission failure can now be quantified. The probability density function is given in equation 2.2.15 by the negative exponential distribution for a component, \(A\), with a constant failure rate in a non-repairable single phase mission.

\[ f(t) = \lambda_A e^{-\lambda_A t} \] (2.2.15)

It is assumed that component \(A\) has a constant failure rate for all phases of a mission. This is regardless of whether it is required for a certain phase or not. To model the unreliability of component \(A\), \(q_A(t)\), over a duration \([0, t)\) a cumulative probability function \(F_A(t)\) is used, as seen in equation 2.2.16.

\[
q_A(t) = F_A(t) = \int_0^t f_A(t) dt = \left[-e^{-\lambda_A t}\right]_0^t = 1 - e^{-\lambda_A t}
\] (2.2.16)

The unreliability of the component over a phase \(i\) is derived in a similar way as equation 2.2.16, as seen in equation 2.2.17. This equation calculates the probability density function for the time of phase \(i\)’s, i.e. phase \(i\) duration is between \(t = t_{i-1}\) and \(t = t_i\).

\[
q_{A_i}(t) = \int_{t_{i-1}}^{t_i} f_A(t) dt = \left[-e^{-\lambda_A t}\right]_{t_{i-1}}^{t_i} = e^{-\lambda_A t_{i-1}} - e^{-\lambda_A t_i}
\] (2.2.17)

For each phase \(i\), the unreliability, \(Q_i\), is found using the inclusion-exclusion expansion for the existence of prime implicants, \(K_{i1}\), in phase \(i\).
2.2. Non-Repairable Systems

\[ Q_i = \sum_{l=1}^{N_{pi}} P(K_{il}) - \sum_{l=2}^{N_{pi}} \sum_{n=1}^{l-1} P(K_{il} \cap K_{ni}) + \cdots + (-1)^{N_{pi}-1} P(K_{1l} \cap K_{2l} \cdots \cap K_{N_{pi}}) \]  

(2.2.18)

Where \( N_{pi} \) is the number of prime implicant sets in phase \( i \).

Using the three-phase mission example in Figures 2.4-2.6, the inclusion-exclusion expansion can be applied to each phase, yielding equations 2.2.19, 2.2.20 and 2.2.21 (phase 1, 2 and 3 respectively).

\[ Q_1 = q_{A1} q_{B1} \]  

(2.2.19)

\[ Q_2 = (1 - q_{A1}) q_{C12} q_{D12} + (1 - q_{B1}) q_{C12} q_{D12} \]  

(2.2.20)

\[ Q_3 = (1 - q_{A1})(1 - q_{C12}) q_{B1} q_{D13} + (1 - q_{A1})(1 - q_{C12}) q_{B2} q_{D12} + q_{B2} q_{D3} + q_{B3} q_{D3} \]  

(2.2.21)

To calculate the mission unreliability, \( Q_{MISS} \), equation 2.2.22 can be used as each phase failure is mutually exclusive and therefore can be expressed as the sum of all the individual phase failures.

\[ Q_{MISS} = \sum_{i=1}^{m} Q_i \]  

(2.2.22)

Where, \( m \) is the total number of phases.

2.2.1.4 Importance Measures for Phased-Mission Systems

Andrews (2008) gives a detailed description of the importance measures of component contribution to the failure of phased-missions. The first to consider is the in-phase criticality function, which is an extension on the Birnbaum’s measure of importance, \( G_i \). Equation 2.2.23 gives this equivalent measure where \( G_{ij} \) is the probability that the system resides in a critical condition such that if component \( i \) fails during phase \( j \), the system would fail. \( Q_i \) is the value that is calculated from the minimal cut sets using the method discussed earlier by La Band. By using that method of obtaining the failure probability of each phase, the success of previous phases is accounted for, i.e. successful transition to the current phase.

\[ G_{ij} = \frac{\partial Q_j}{\partial q_{ij}} \]  

(2.2.23)
There are two criticality importance measures that need to be considered for phased-mission systems: *Phase Importance* and *Transition Importance*. These can both lead to phase failure and therefore mission failure. Each is given in detail below:

- **Phase Importance**: Within any phase, a system can be in a critical condition for component \( i \) in phase \( j \), and phase failure can occur due to component \( i \) failing. Equation 2.2.24 is the fraction of phase failures that occur due to component \( i \) failing.

\[
I_{ij}^P = \frac{G_{ij}q_{ij}}{Q_j} \quad (2.2.24)
\]

- **Transition Importance**: If the failure conditions of phase \( j \) are met before entering this phase, given that all previous phases are successful, phase failure will occur on transition to phase \( j \). Equation 2.2.25 is the proportion of the total phase failure probability that component \( i \) contributes to cause the transition failure into phase \( j \).

\[
I_{ij}^T = \frac{\left( \sum_{k=1}^{j-1} G_{ij,k}q_{ik} \right)}{Q_j} \quad (2.2.25)
\]

The total importance contribution of component \( i \) failure in phase \( j \) is found by summing the contribution from transition importance and the phase importance as shown in equation 2.2.26.

\[
I_{ij} = I_{ij}^P + I_{ij}^T \quad (2.2.26)
\]

The total contribution by component \( i \) to the mission failure can be calculated using equation 2.2.27, which is the proportion of mission failures that triggers component \( i \) to fail, given that the system was in a critical state for component \( i \).

\[
I_i = \frac{\sum_{allj} \left\{ \left( \frac{\partial Q_j}{\partial q_{ij}} \right) q_{ij} + \left( \sum_{k=1}^{j-1} \left( \frac{\partial Q_j^T}{\partial q_{ik}} \right) q_{ik} \right) \right\}}{Q_{MISS}} \quad (2.2.27)
\]

### 2.2.2 Phase Modular Approach

The initial properties of modules and their use in fault trees are described in Chatterjee (1975) and Birnbaum and Esary (1965). Chatterjee describes two properties of modules in fault trees as the following:
2.2. Non-Repairable Systems

1. All branches of the fault tree are independent.

2. The logic function associated with each gate, is one of the following:
   - A prime. In this representation a prime function is any gate other than those with a AND or OR function.
   - An AND with no other inputs immediately below it that are also AND gates.
   - An OR with no other inputs immediately below it that are also OR gates.

Birnbaum and Esary give a definition of a module in coherent systems. A module of a system is a subset of basic components of the system which are then organised in to their own substructure. When organised in to these substructures they can be treated as a component of the system. Components only affect the system through the performance of their substructure. These lead to Locks (1981) expanding this idea for non-coherent fault trees and showed how it could be used to find the cut sets.

Meshkat et al. (2003) discuss a method for modularising fault trees to represent phased-mission systems using combinatorial and Markov-chain based methods. Within each phase there are interdependencies between the components, which are brought together. The phase modular approach identifies modules within the fault trees of the phased mission that remain independent throughout. An example of this can be seen in Figure 2.8 where the modules of the tree have been identified. These modules do not exist anywhere else in the tree, therefore are independent.

The reliability of each of these modules is found and then these modules are combined in a system level BDD. This is used to find the system reliability measures. The process
involves finding independent modules within a fault tree and then solving each, depending on whether the phase is static or dynamic and then integrating the two methods to obtain the reliability of the system. This process is referred to as modularisation.

One type of combinatorial method for a phase modular approach is the Binary Decision Diagram (BDD) approach, which is computationally efficient. However to use this, every phase of the mission would be required to be a static phase. For any dynamic phases the Markov chain models can take care of any dependencies between the components and the order in which they can fail. Both methods discussed here do have a common limitation in that they assume statistical independence among the failures of the components.

The phase modular approach starts by using fault trees to represent each of the phases of the mission. Sub-trees are created to identify the independent components in the phase. Each of these sub-tree is identified as either static or dynamic. The system-level independent modules, are those that overlap in at least one component. For example one module could contain \{A, B, D, E\}, another \{A, B, D\} and another \{D, E\}. The system-level independent module would be \{A, B, D, E\}. The modules would then be stated as either static or dynamic. Static modules would include all AND, OR and/or K-out-of-n gates. Dynamic modules would include at least one of the following; priority AND (PAND), cold spare (CSP), warm spare (WSP) or hot spare (HSP) gates. Once this has been established each of the modules is identified as being bottom-level or upper-level. Bottom-level means that the module has no child modules and the top-level means that the module does have child modules. An example of a bottom-level module can be seen in Figure 2.8 is module \(M_1\), and an example of a top level module is \(M_3\) as it has child modules \{A, B\} and \{B, E\} which are each linked to a gate.

Once all bottom-level modules have been identified, the BDD approach is used to find the joint phase module probabilities if the modules are static, and if the modules have dynamic properties then the Markov approach is implemented. As the reliability measures were found, each of the modules were treated as a basic event of a static fault tree. Through the reliability measures found earlier, the system reliability equation can be found by solving the BDD from that static fault tree.

An algorithm to detect modules within fault trees is described by Dutuit and Rauzy (1996). The algorithm is based on Tarjan’s (1972) algorithm, which describes a method for depth-first search and linear graphs. Dutuit and Rauzy describe an algorithm that finds strongly related components of a graph. The algorithm presented was designed for large fault trees with several hundred gates and events.

Another phase modular approach from Ou and Dugan (2004) is for dynamic multi-phase systems. Two concepts were introduced; the first is the phase module and the second is module joint probability. A module joint probability is defined by Ou and Dugan as ‘the probability of a module with the specified statuses, either operational or failed, in different
Table 2.2: Phase algebra used by Zang et al. (1999) \((i < j)\)

<table>
<thead>
<tr>
<th>Algebraic Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\overline{A}_i \cdot \overline{A}_j \rightarrow \overline{A}_j)</td>
</tr>
<tr>
<td>(A_i \cdot A_j \rightarrow A_i)</td>
</tr>
<tr>
<td>(A_i + A_j \rightarrow A_j)</td>
</tr>
<tr>
<td>(A_i + A_j \rightarrow A_i)</td>
</tr>
<tr>
<td>(A_i + A_j \rightarrow 1)</td>
</tr>
</tbody>
</table>

phases’. For example the module joint probability of \(P\{M_1 M_2\}\), where \(M_{1,j}\) is module one in phase \(j\), this is the probability that module \(M_1\) is operational in phase 1 and failed in phase 2. From this method the reliability of the multi-phase system, i.e. that it is operational throughout the mission, can be calculated by doing a special joint probability of all the phases in the mission. There are two ways in which this is found; the first is based on basic events only, i.e. the module has no child sub-modules. The second depends on one or more Module Basic Event (MBE), i.e. the module is not at the bottom-level of the system, and has child sub-modules. The only disadvantage in using the joint probability method is that it can increase the computational time, and decrease efficiency.

2.2.3 Binary Decision Diagrams for Phased-Mission Systems

Zang et al. (1999) designed an algorithm that would use BDDs to analyse phased-mission systems. Phase algebra is used to cover the dependencies across the phases, with a new operation for a BDD to incorporate this phase algebra. The phase algebra used for this technique can be seen in Table 2.2.

The failure function of a component in a specific phase is given by equation 2.2.28. This takes care of the statistical-independencies across the phases. Equation 2.2.28 is the failure function for component \(C_A\) in phase \(j\), with the time period, \(0 \leq t \leq T_j\).

\[
F_{A,j}(t) = \left[1 - \prod_{i=1}^{j-1} (1 - p_{A,i}(T_i))\right] + \left[\prod_{i=1}^{j-1} (1 - p_{A,i}(T_i))\right] \cdot p_{A,j}(t) \quad (2.2.28)
\]

Where \(p_{A,i}(t_i)\) is the failure function of \(c_{A_i}\). This has been defined below:

\[
p_{A,i}(t_i) = \begin{cases} 
Pr\{A(t) = 0\} & i = 1 \\
Pr\{A(t + T_{i-1}) = 0|A(T_{i-1} = 1))\} & 1 < i \leq j; t \leq T_i
\end{cases}
\]

Where, \(A(t)\) is the state indicator variable for \(C_A\) and \(T_i\) is the duration of phase \(i\).

The first term in equation 2.2.28 represents the probability that a component has failed in the previous phase \((1, 2, ..., j - 1)\) and the second term represents the lifetime probability distribution of the component in phase \(j\).
The BDD algorithm uses the phase algebra stated in Table 2.2 to create a new BDD operation, Phased-Dependent Operation (PDO). PDOs come in two forms; forward and backward PDO. This is due to the dependence of BDD structures on ordering.

- **Forward PDO**: The order of the variables is the same as the phase order,
  
  \[ A_1, A_2, \ldots, A_n \]

- **Backward PDO**: The order of the variables is in reverse to the phase order,
  
  \[ A_n, A_{n-1}, \ldots, A_1 \]

Where \( A_i \) is the state indicator variable of component \( A \) in phase \( i \).

Using the \texttt{ite} structure technique, \( E_i \) and \( E_j \), representing the failure combinations for phase \( i \) and phase \( j \) respectively, can be represented as follows:

\[
E_i = \texttt{ite}(A_i, G_1, G_2) \tag{2.2.29}
\]

\[
E_j = \texttt{ite}(A_j, H_1, H_2) \tag{2.2.30}
\]

Where \( H_1, H_2, G_1 \) and \( G_2 \) are the \texttt{ite} structure off the branches.

For a forward PDO,

\[
\texttt{ite}(A_i, G_1, G_2) \oplus \texttt{ite}(A_j, H_1, H_2) = \texttt{ite}(A_i \oplus H_1, G_2 \oplus E_j) \tag{2.2.31}
\]

If \( A \) has failed in phase \( i \) then \( A \) must be in a failed state in phase \( j \).

For a backward PDO,

\[
\texttt{ite}(A_i, G_1, G_2) \oplus \texttt{ite}(A_j, H_1, H_2) = \texttt{ite}(A_j \oplus E_i \oplus H_1, G_2 \oplus H_2) \tag{2.2.32}
\]

If \( A \) is working in phase \( j \) then it must have been working in phase \( i \).

The ordering strategy for ordering variables is based on a heuristic. Using this information the components can be ordered. The heuristic is where each basic event of the fault tree is given the value of weight 1, and therefore the weights of the gates can be found by adding the weights of all inputs. When the whole tree has been assigned weights, a depth-first traversal of the tree is made. At each level the sons of a gate are chosen by order of increasing weight. During this process of traversal, as soon as variables are encountered they are put in an ordered list.

Once ordered, each component indicator variable was then replaced with a set of variables representing the component in each phase by using one of the above two PDOs.
2.2. Non-Repairable Systems

(Forward or Backward), where each has its own ordering of the variables that belong with the same component.

The process of the BDD algorithm using backward PDO can be seen below:

1. The failure function for each variable is calculated using equation 2.2.28.

2. The components and their corresponding variables are ordered using the heuristic described earlier.

3. For each phase a BDD is generated using ordinary logical operations, as seen in §1.4.1.3.

4. Using the phase algebra stated in Table 2.2 and the corresponding backward PDO the final BDD can be found by combining these.

5. The unreliability of the PMS is calculated from the final BDD. This is accomplished using an evaluation algorithm.

The results of the algorithm found that the backward PDO created a smaller BDD. The reason for generating a smaller BDD was that the backward PDO provides the advantage of cancelling common components automatically.

Xing and Dugan (2004) commented on the algorithm by Zang et al. (1999), and provided areas of improvement to the algorithm. Two new rules were suggested by Xing and Dugan to overcome an unstated restriction on the variable ordering. The PDO presented by Zang et al. (1999) was used to combine single-phase BDDs to obtain the PMS BDD. When the same component root node occurs in different phases, the PDO would be applied to the combination rather than the usual BDD operation which would address the statistical-dependency that exists between the variables of the same component in different phases when combining them. The two rules state:

- **Rule 1**: Orderings adopted in the generation of each phase BDD are consistent or the same for all the phases.

- **Rule 2**: Orderings of variables that belong to the same component, but to different phases, stay together. This is achieved by replacing each component indicator variable with a set of variables which represent this component in each phase after the ordering of components is completed using the heuristic stated earlier.

If the rules are not followed correctly would lead to inaccurate single-phase BDDs that are used to generate the final PMS BDD.

La Band and Andrews (2004) discuss a method of using BDDs to represent phase fault trees. In Section 2.2.1 details of using just fault trees were given. This is extended to
convert the fault trees into BDDs in order to increase the efficiency of the mathematical manipulation. Although BDDs are very difficult to generate directly from a system description, fault trees are not; but when a system is large it is more efficient to convert it.

Each phase of the mission is represented by a phase fault tree and then converted into a BDD. Each component occurs in the BDD corresponding to the phase being evaluated. These BDDs are evaluated in the usual way, by finding the paths to the ‘1’ terminal nodes. To account for the phase dependencies, the laws given in Table 2.1 are used. This is a simple and effective method of evaluating a phased-mission system. The advantage of using BDDs over the fault tree method come when completing the mathematical analysis. It is more advantageous to convert fault trees into BDD form, particularly for larger systems as they can produce large and complex fault trees. This is also true for non-coherent fault trees such as phase failure fault trees.

Dunnett and Andrews (2006) discuss a method that uses BDDs for non-repairable phased-mission systems. As before, the phase fault trees are converted into their equivalent BDD form which is expanded to include all components that appear in subsequent phases. This method uses these BDDs to create the mission BDD. To minimise the size of the mission BDD the following rules are used:

- **Rule 1**: $A_i \cdot A_j = 0$ Non-repairable component $A$ cannot fail in phase $i$ and $j$.
- **Rule 2**: $A_i \cdot \bar{A}_j = 0$ for $i < j$, Component $A$ cannot be repaired once failed.
- **Rule 3**: If $C_1$ is a minimal cut set for phase $j$ and that cut set occurs prior to entering phase $j$, then the system will fail when it enters phase $j$.

These rules are applied to produce a reduced mission BDD. The phase and mission failure probability can be calculated by first identifying the paths that lead to a phase failure. Using phase algebra these paths can be minimised and evaluated. The phase failure probability is found by summing the probabilities of all the paths. A general algorithm for the construction of a mission BDD is given below:

1. Using the constructs for AND and OR gates to produce a BDD for each phase of the mission from the phase fault trees. These are expanded to incorporate all possible states of components required in later stages.
2. Considering the mission failure as an OR combination of the phase failures, the mission BDD is constructed. Terminal mission BDD nodes represent the phase in which the mission fails or succeeds. As there are more than two possible outcomes, this BDD differs from conventional BDDs.
3. Each phase failure BDD is incorporated one at a time into the mission BDD to yield a structure that is defined by the performance of each component in each phase.
4. The mission BDD is minimised by removing paths which represent impossible component conditions.

The general algorithm for mission BDD quantification is given below:

1. Failure modes for each phase of the mission are obtained by considering the component conditions represented by each path of the BDD leading to the specified phase failure.

2. Failure modes are simplified by using the phase algebra.

3. Component phase failure probabilities are evaluated.

4. Phase failure likelihoods are calculated using the disjoint phase failure modes and the component failure probabilities.

5. Phase failure probabilities can be combined with the consequences of phase failure in order to perform a mission risk analysis, or summed to find the mission failure probability.

Prescott et al. (2009) discuss a method where BDDs can be used for phased-mission planning. This work was based around phased-mission planning of autonomous systems, which require quick analysis in order to enhance the capabilities of the autonomous system decision-making. There are four stages to the methodology. The first is the phase failure logic BDD construction denoted $F_i$. The phase failure for every possible phase during a mission is represented by fault trees, which are then converted to BDDs. The BDDs are independently structured to allow for an ordering scheme to be chosen, which will minimise the size of the BDD. Once constructed the BDDs can be saved in a library for later use.

The second stage is the mission definition. The profile of the mission, the order of the tasks and time taken for each phase are decided in this step. The appropriate BDDs are then chosen from the library. These BDDs represent $F_i$ which would be used to make up the mission failure of every phase, denoted, $Ph_i$. In the third step of this technique, the quantitative analysis begins. For the construction of the BDDs that represent $Ph_i$, a simple connection process is used. This process does not require variable ordering and allows efficient connection of the $F_i$ BDDs. These have their own variable ordering. This helps to minimise the size of the BDD. This process uses the success of the mission, which switches the terminal nodes 0 and 1 of the BDD. This gives the dual BDD representing the success of a phase. When the $Ph_i$ BDDs are built, the AND connection of two BDDs is achieved by connecting all terminal 1 nodes of one BDD to the root node of the other BDD to be connected. As different BDDs might have identical components usually these would have to follow a specific ordering scheme which would cover both BDDs. This method, however,
does not have to as when the variables are connected they are treated independently, with the times associated with the variables are used to take care of the dependencies between them during quantification. By using this method the time to construct a mission phase failure BDD is reduced. The fourth, and final step entails the quantification of the $Ph_i$ BDDs. The failure probabilities are calculated and then processed for their viability.

2.3 Repairable Systems

2.3.1 Markov applications in Phased-Mission Systems

For repairable systems the assumption of independence is no longer valid, in this case the Markov approach is valid. There has been some work in using various Markov models to represent phased-mission systems. Clarotti et al (1980) discusses a method that uses the Markov approach to represent a repairable system undergoing a phased mission. This method takes in a reliability model, such as a reliability block diagram and phase state description. To demonstrate the method presented, an example system will be used. In Figure 2.9, the RBDs for each phase are given, and in Table 2.3 the phase state description is given. This table consists of all possible states the system can exist in, listing each component as either in a functioning state (0) or failed state (1). The phase time periods (start time, end time) are as follows:

- Phase 1: $(0, t_1)$
- Phase 2: $(t_1, t_2)$
- Phase 3: $(t_2, t_3)$

Starting with phase 1, a probability vector, $P(0)$, is used to state what the probability is of the system existing in each of the eight states at time 0. It is assumed at the beginning
of the mission that all components are functioning, therefore the probability of the system existing in state 1 at time 0 is 1, as shown in equation 2.3.1.

\[ P(0) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \] (2.3.1)

From the system configuration shown in Figure 2.9a, either A or B or C must be functioning for phase 1 to be successful; therefore the system cannot exist in state \( S_8 \). The progression through phase 1 can be represented in matrix form as seen in equation 2.3.2, which comes from the matrix equation 1.4.42.

\[
\begin{bmatrix}
\dot{p}_1(t) \\
\dot{p}_2(t) \\
\dot{p}_3(t) \\
\dot{p}_4(t) \\
\dot{p}_5(t) \\
\dot{p}_6(t) \\
\dot{p}_7(t) \\
\dot{p}_8(t)
\end{bmatrix} =
\begin{bmatrix}
-\sum_1 & \nu_A & \nu_B & 0 & \nu_C & 0 & 0 & 0 \\
\lambda_A & -\sum_2 & 0 & \nu_B & 0 & \nu_C & 0 & 0 \\
\lambda_B & 0 & -\sum_3 & \nu_A & 0 & 0 & \nu_C & 0 \\
0 & \lambda_B & \lambda_A & -\sum_4 & 0 & 0 & 0 & - \\
\lambda_C & 0 & 0 & 0 & -\sum_5 & \nu_A & \nu_B & 0 \\
0 & \lambda_C & 0 & 0 & \lambda_A & -\sum_6 & 0 & - \\
0 & 0 & \lambda_C & 0 & \lambda_B & 0 & -\sum_7 & - \\
0 & 0 & 0 & \lambda_C & 0 & \lambda_B & \lambda_A & -
\end{bmatrix}
\begin{bmatrix}
p_1(t) \\
p_2(t) \\
p_3(t) \\
p_4(t) \\
p_5(t) \\
p_6(t) \\
p_7(t) \\
p_8(t)
\end{bmatrix}
\] (2.3.2)

Where, \( \sum_i \) is the sum of the ith column, 0 represents impossible state transitions and ‘-’ represents absorbing states.

To move from phase 1 to phase 2 at time \( t_1 \), the system must exist in states that allow success for both phase 1 and phase 2. To be successful in phase 2, A must be functioning with either B or C functioning. Therefore, the system can exist in \( S_1, S_3 \) or \( S_5 \). The probability that the system will exist in one of these states at time = \( t_1 \) (moving from phase 1 to phase 2) is the sum of the probability of being in each state as shown by equation 2.3.3.

\[ R(t_1) = P_{S_1}(t_1) + P_{S_3}(t_1) + P_{S_5}(t_1) \] (2.3.3)
Where $R(t_1)$ denotes the reliability of the system at time $t_1$.

For phase 2 ($t_1$, $t_2$), the system must start this phase in either $S_1$, $S_3$ or $S_5$ as discussed above. At this point all other states are said to be absorbing, as entering any of the other states will cause mission failure. Therefore the initial probability vector is given as follows:

$$P(t_1) = \begin{bmatrix} P_{S_1}(t_1) & 0 & P_{S_3}(t_1) & 0 & P_{S_5}(t_1) & 0 & 0 & 0 \end{bmatrix}^T$$ (2.3.4)

The matrix equations for phase 2 are given in equation 2.3.5.

$$\begin{bmatrix} \dot{p}_1(t) \\ \dot{p}_2(t) \\ \dot{p}_3(t) \\ \dot{p}_4(t) \\ \dot{p}_5(t) \\ \dot{p}_6(t) \\ \dot{p}_7(t) \\ \dot{p}_8(t) \end{bmatrix} = \begin{bmatrix} -\sum_1 & - & \nu_B & 0 & \nu_C & 0 & 0 & 0 \\ \lambda_A & -\sum_2 & 0 & - & 0 & - & 0 & 0 \\ \lambda_B & 0 & -\sum_3 & - & 0 & 0 & - & 0 \\ 0 & \lambda_B & \lambda_A & -\sum_4 & 0 & 0 & 0 & - \\ \lambda_C & 0 & 0 & 0 & -\sum_5 & - & - & 0 \\ 0 & \lambda_C & 0 & 0 & \lambda_A & -\sum_6 & 0 & - \\ 0 & 0 & \lambda_C & 0 & \lambda_B & 0 & -\sum_7 & - \\ 0 & 0 & 0 & \lambda_C & 0 & \lambda_B & \lambda_A & - \end{bmatrix} \begin{bmatrix} p_1(t) \\ p_2(t) \\ p_3(t) \\ p_4(t) \\ p_5(t) \\ p_6(t) \\ p_7(t) \\ p_8(t) \end{bmatrix}$$ (2.3.5)

For phase 2 to be successful the system must exist in a state that is successful for both phase 2 and 3. For success in phase 3 as shown in Figure 2.9c A, B and C must all be working for success in phase 3. Therefore the system must exist in state, $S_1$ at time $t_2$. The probability that phase 2 is successful is therefore given as equation 2.3.6.

$$R(t_2) = P_{S_1}(t_2)$$ (2.3.6)

For the third and final phase of the mission, the system must be in state $S_1$ at time $t_3$ in order for the phase to be successful. The probability vector at time $t_2$ is given in equation 2.3.7.

$$P(t_2) = \begin{bmatrix} P_{S_1}(t_2) & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$ (2.3.7)

The matrix equation for phase 3 is given in equation 2.3.8.
The probability that phase 3, and therefore the mission is successful is the probability that the system remains in state $S_1$ until time $t_3$, the end of the mission. This is given in equation 2.3.9.

$$R_{\text{MISSION}} = P_{S_1(t_3)}$$ (2.3.9)

This method moves from one phase to another accounting for the requirements to move between phases. The phase reliabilities are based on what possible states the system can exist in to make a phase successful, and the final phase provides the overall reliability of the mission.

Alam and Al-Saggaf (1986) discuss the reliability of repairable phased-mission systems. The analytic Markov model presented by Alam and Al-Saggaf solves for systems where the Mission-Phase Change Times (MPCT) are deterministic, and further to stochastic (only the deterministic MPCT method is covered here). There are two methods presented, the difference between the two is in determining the sequence of initial conditions for subsequent phases. This means that the phases would be assessed individually as soon as the initial conditions are set by the previous phase. For random MPCTs, the marginal distribution of the duration of each phase has to be determined as well, in order to find the initial conditions for the next phase. Five assumptions were given that apply to this work as follows:

1. A system has both good and bad elements. A mission (phase) requires several such elements.

2. Failure and repair times are statistically independently exponentially distributed.

3. Repaired items are always returned to a state of as-good-as-new.

4. If the system fails during a particular phase the process ends.

5. Transition time is instantaneous between any consecutive phases in the mission.
For deterministic MPCTs, each phase can be evaluated by identifying all possible states the system can be in, e.g. for a system with components A, B, C, there are eight possible states. Three components with two possible states each leads to $2^3 = 8$ possible states. This failed state would be an absorbing state. The state transition matrix $[A]$, can be found by the information given above. If a component fails then the repair on that component starts immediately.

Using equation 1.4.42 with the initial condition that all components are working as $P(0)$, given in equation 2.3.1, The probability of being in each state at the end of the phase is determined.

In each phase a number of components are required to be working for a phase to be successful. This method considers the state of components in the current phase and the subsequent phase. The reason it considers subsequent phases is that one (or more) component(s) may need to be working in both the current and the subsequent phase in order to move from the current phase to the subsequent phase. From the initial conditions set in equation 2.3.1 and the state transition matrix discussed earlier, the reliability of the system can be found. This is the probability of being in one of the states that has all the required components in a working condition. For all subsequent phases after the first the same approach is taken, but the initial conditions are set by the reliability equation. The success of the mission is dependent on the reliability found at the end of the last phase. This says what states must be occupied in order for the mission to be a success.

Smotherman and Zemoudeh (1989) use non-homogeneous Markov models for reliability analysis of phased-mission systems. Three assumptions that hindered previous work, in that they restrict the flexibility and applicability of the work, were considered:

- Phase changes and phase-change times do not depend upon individual states, but rather only upon the current phase. It is therefore not possible to represent a degraded system which requires longer completing a phase than a fully working system.

- The number of phases of random duration must be restricted, or the time-in-phase of each phase of random duration must obey an exponential or locally-time-dependent distribution.

- Failure and repair rates must be constant within phases. Constant failure and repair rates have been used to model the useful life period of components, which does not take into account the burn-in phase of electric components and the wear-out effects of mechanical components.

Smotherman and Zemoudeh (1989) took each of these and produced a model that can have phase-change times that are dependent and phases that have random durations.
2.3. Repairable Systems

represented by globally-time-dependent distributions of phase-change times. The model also states that the failure and repair rates are globally-time-dependent. This model assumes that the system can be represented by a continuous-parameter finite-state Markov model, with non-overlapping uniform distributions for phase-change times and failure and repair rates globally-time dependent, where the repairs are modelled as “continuous wear”. A set of sub-states represent each phase of a mission and the transitions are generalised to handle phase changes. Also the transitions are given in terms of random variables which satisfies the above hindrances. The only disadvantage to this technique is that the Markov model generated can be very large. This is computationally inefficient, and can have an effect on the type of systems that could be modelled.

Dugan (1991) gives a method to automate the analysis of phased-mission systems. The method is based on Markov models created from fault trees that represent the phases of a defined mission. In this model the phase change times are fixed; random phase changes were not considered at this point. The model is similar to Smotherman and Zemoudeh in that a single model is created to represent all the phases of the system.

Some of the methods for analysing phase-mission systems, including some of the above are computationally inefficient. Somani et al. (1992) describes a technique to increase the efficiency with which phased-mission systems can be analysed. This technique takes into account systems with variable configurations, spares within the systems either for a single component (dedicated spare), or for an array of components (pooled spare). These spares can be brought into service to aid in balancing reliability and costs depending on the requirements for the given phase. The technique incorporates redundancy management in the system into the Markov model. This is done by generating a Markov model for every phase, rather than a single Markov model. These Markov models are then solved independently. Mean Time Between Critical Failures (MTBCF) is presented for a mission, \( M \), of time, \( T \), seen in equation 2.3.10. The MTTF for a mission is given in equation 2.3.11.

\[
M(T) = \int_0^T \frac{R(t) \, dt}{1 - R(T)} \quad (2.3.10)
\]

\[
\mu = M(\infty) \quad (2.3.11)
\]

An equivalent probability of occupation for a virtual state \( S \) for a time \( t \) is given as equation 2.3.12. The transitions are also virtual and the transition rate from all of the operating states to this new state is 1.

\[
S(T) = \int_0^T R(t) \, dt \quad (2.3.12)
\]
After some manipulation the MTBCF can be calculated using equation 2.3.13. At the end of the mission both $\bar{S}(T)$ and $Q(T)$ are known and $\bar{S}(T)$ is calculated using the virtual state.

$$MTBCF = \frac{\bar{S}(T)}{1 - Q(T)}$$ \hspace{1cm} (2.3.13)

Kim and Park (1994) describe a technique that uses Markov-based applications to assess the reliability of a multi-phase mission systems where the configuration of the system changes during consecutive time periods. This assumes that the failure and repair rates of the components in a system are exponentially distributed, and that components of a redundant nature are repairable, given that the system is operational. Kim and Park discuss three cases in which to apply their Markov model; phase durations that are deterministic, and random variables with either a set maximum mission time or no set mission time. This technique uses system eigenvalues to solve the differential equations created from the Markov model. These eigenvalues are found in the reduced TRM, $[B]_k$, for the $k^{th}$ phase. For each of the cases, the TRM with all input and outputs are stated and from this the reduced TRM is given for each of the phases. The eigenvalues are then calculated for each of the phases, which is used to find the mission reliability.

Alam et al. (2006) discuss an approach that uses both Markov discrete and continuous models, by changing the phased-mission system into separate non-phased-mission systems. The Markov model uses a standard Markov chain resulting in the union of $2^n$ possible states encountered during the mission. The Markov model can then be used to calculate probability of mission success and the MTTF. Some of the advantages of this method is that once the models are created any decisions made before or during a mission can be done without having to re-calculate the models. Another advantage is that every phase is calculated independently, which makes it quick and simple to input into a program. A major advantage is that as all the phases are self-contained this allows the user to change the number of phases and duration for a particular mission. Also, should the sub-mission requirements change, the reliability of the mission can be calculated easily.

As there is an issue with state explosion with Markov models for real-world applications, this method bypasses this problem by remembering that the overall state space of the mission is the union of all the states encountered in all the phases. The procedure for this technique is given as the following:

1. Calculate the reliability of the $m$ phases from $t_0$ to $t_f$, covering all anticipated durations for each phase.

2. The reliability for the phased-mission system is found by selecting one of the applicable $m$ phase durations. Initial and final probabilities for phase $i$ for a specific
duration are maintained at time \( t_{i-1} \) and \( t_j \), respectively.

3. The worst-case estimate (or conservative estimate) is calculated when the individual reliability values for the \( m \) phases are multiplied together.

### 2.3.2 System and Phase Petri Nets

Mura and Bondavalli (2001) describe a method of modelling Phased-mission systems as a combination of two separate models. These models are called System Net (SN) and Phase Net (PhN). A System Net is used to represent an overview of all the components in the system, taking into account their interactions and their failure and repair behaviour. The Phase Net is used to describe the changes between phases throughout the mission. Chew et al (2008) extends this method by using three different nets, as opposed to the two proposed by Mura and Bondavalli. These three nets are; Phase Petri Net (PPN), Component Petri Net (CPN) and Master Petri Net (MPN). The PPN describes the phase failure of the system in terms of the components or basic event failures. The CPN describes the failure/repair characteristics for each component. In this model the components can be repaired at the end of each mission, which is made up of a number of phases. The MPN controls the systems progress through the phases, whether it has failed in a phase and hence cannot progress or it is allowed to progress through the phases. In this study by Chew et al. it also controls the maintenance of the components. The Petri nets described above interact through arcs and transitions. Together the PPN, CPN and MPN create one large Petri net.

The Phase Petri net shows the system failure in terms of the basic components. It is a Petri net representation of the phase fault tree.

The Component Petri net shows the basic event failure, this includes the time for Maintenance. Each component is graphically represented by two places in the Petri net; one to show it is in a working state, the other shows the failed state of the component. The number of places are not limited to show just working and failed, there are other states in which the component can reside. These are then linked to the Phase Petri net from the failed state place and to the Master Petri net if enough of the components could cause mission failure. Each of the components are also linked to an area of the CPN that denotes the repair place for the components.

The Master Petri net consists of three main sections. The first is the area for control of the sequence of phases, and the failure/success of the mission. In terms of the control of the phases, this section of the Master Petri net indicates with a token which phase is currently in operation. Should a Phase failure occur it would be indicated on the Phase Petri net as the top event and indicated in the Master Petri net that it is a failed phase. If that phase is also the current operational phase then the mission would fail. This type of phase
modelling was adapted from the work by Volovoi (2004). The second is the ending of each mission or maintenance free operating period (MFOP). Should a mission be accomplished, then one of two actions occur next; another mission is initiated before any maintenance occurs or the system enters a maintenance recovery period (MRP). This is where any repairs needed are taken into account. The last is the section that takes into account the abandonment of the mission should a component or system fail. This is indicated on the Master Petri net with a place indicated as the Mission abandoned. Should a component or system fail that causes a mission abandonment, all missions cease and the maintenance free operating period is considered failed and the system enters a maintenance recovery period.

Chew et al. (2008) developed simulation software to find the overall system reliability of a system. As input the software takes the phase fault trees and component failure data in a text-file format. The software then takes this information to produce the three Petri nets; MPN, PPN and CPN. The output for this software is again in the format of a text-file that can then be used for analysis. The simulation itself takes the component failure data and phase lengths and uses these to give the transition times required. These times and the component failure data and the phase lengths can all be sampled using distribution types such as normal and exponential. The algorithm used for this simulation is given below:

1. Randomly sample switching times for each newly enabled timed transition in each net from the switching time distribution assigned to it.

2. Find the transition with the earliest switching time and fire it.

3. Search through each of the immediate transitions and if any are enabled, switch them.

4. Repeat step 3 until no more immediate transitions are enabled.

5. Test for any of the following conditions and log them:

   (a) If a system has failed, begin next simulation
   (b) If system has been abandoned, begin next MFOP
   (c) If a mission has completed begin next mission
   (d) If MFOP has completed, begin MRP
   (e) If MRP has completed, begin next MFOP
   (f) If simulation has completed, begin next simulation

6. If simulation completed < $n_s$, go to step 1, else end.
2.4 Summary

The reliability modelling for Phase Mission Systems showed a variety of ways to model with the methods discussed in Chapter 1. The methods discussed there were applied to non-repairable and repairable systems. Ideally the method to be taken forward would have to cater for both repairable and non-repairable systems, without the need of combining with another method. From the research, the method from Chew et al. (2008) which was the development of a piece of software to find system reliability using Petri nets as the modelling method had the most potential. The method in which the Petri nets are arranged provides the foundation of the work presented here.
Chapter 3

Automated Techniques

Contents

3.1 Introduction .................................................. 73
3.2 Methods for Automation of Reliability Models .................. 73
  3.2.1 Decision Table Methods .................................. 73
  3.2.2 Digraph Method ......................................... 77
  3.2.3 Modified Decision Table method ......................... 77
  3.2.4 Cause-Consequence Diagrams ......................... 79
  3.2.5 Mini fault trees ......................................... 79
  3.2.6 Faultfinder ............................................. 79
3.3 Summary .................................................... 85

3.1 Introduction

The ability to automate a reliability method is not a new concept. Most effort has
been in the development of the automated construction of fault trees. Manual fault tree
construction is inefficient and prone to error. Although a specialist would be able to
complete this task themselves, there are two reasons that this is inefficient. The first
is the number of man hours required to complete such a task particularly if the system
is large. The second is fault tree construction can be subjective, meaning that different
specialists might adopt inconsistent approaches. To reduce the amount of time that is
needed to construct a fault tree and ensure that the tree is accurate, automated methods
were needed to complete the task. This is the same for any reliability model.

3.2 Methods for Automation of Reliability Models

3.2.1 Decision Table Methods

Decision tables are a method of defining the behaviour of a component within a system.
They can represent the different states, working or failed, of the component and how
the component behaves as a result of different inputs from other components within the
Table 3.1: Complete decision table for component fuse

<table>
<thead>
<tr>
<th>Row No</th>
<th>Input State</th>
<th>Internal Mode</th>
<th>Output State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Input/Output states: 0 - No Signal, 1 - Normal, 2 - High/Overload
Internal mode: 0 - Good, 1 - Failed Open (without an overload input), 2 - Failed shorted (fails to open in the event of an overload)

Table 3.2: Reduced decision table for component fuse

<table>
<thead>
<tr>
<th>Row No</th>
<th>Input State</th>
<th>Internal Mode</th>
<th>Output State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Input/Output states: 0 - No Signal, 1 - Normal, 2 - High/Overload
Internal mode: 0 - Good, 1 - Failed Open (without an overload input), 2 - Failed shorted (fails to open in the event of an overload)

Salem et al (1977) discussed a method using decision tables to model the system behaviour. The decision tables represent the components in the system. For example the decision table for the component fuse has been given in Table 3.1. Decision tables are first created to account for every possible combination of inputs and state and showing the output as a result of the combinations. In the example given in Table 3.1 the inputs, outputs and the internal modes are represented by three values as described in the table.

From this full version of the decision table, modifications can be made in order to reduce its size. This is done by identifying which rows have the same output based on either inputs or states. This allows the rows to merge and therefore reduce the size of the table. An example of this reduction can be seen in rows 1, 2 and 3 where regardless of the internal mode of the fuse if the input state is 0 then the output is 0. By bringing these rows together and introducing ‘–’, which refers to a ‘don’t care’ entry, the table is reduced to that seen in Table 3.2. It should be noted that only the input and internal states can be reduced to a ‘don’t care’ state.
A method of using decision tables to once again model components within a system is demonstrated by Salem et al. (1977). This method discusses how decision tables have been used in aiding the construction of fault trees. A piece of software, Computer Automated Tree (CAT), was developed as a result. The software was designed for the analyst to use as part of their work. The method discussed describes a way of producing an accurate fault tree to describe a system’s behaviour. The software was designed to cater for multiple fault trees at any one time, which could include the success of a system. The CAT allows an analyst to edit the produced fault tree including the gates and events of the fault tree.

Another method for fault tree construction is presented by Han et al (1989) which describes a different piece of software, Automated Fault Tree Construction (AFTC). The AFTC code was designed as an improvement on the CAT software. This method combines the use of decision tables and flow diagrams to ensure that the system and its components can be modelled effectively. This method uses the concept of super component models, which are similar to flow diagrams, together with decision tables and flow diagrams in order to generate a fault tree. Common Cause Failures (CCFs) models, which show how a root cause can create multiple component failures, are then merged into the fault tree. Then the fault tree is modularised. The software holds within a library the decision tables and the flow diagram is a required input from the user. The flow diagram would incorporate the basic decision table of the components. The main feature of this software was the use of these super components and the ability to use CCF modelling and modularising fault trees.

A new method of describing a component’s behaviour was introduced by Majdara and Wakabayashi (2009). They introduce a new table, the state transition table, which describes the operational states of a component. An example of a component with operational states is a switch. This has the operational states of open and closed. Together with the function table is similar to the normal decision tables that have previously been discussed. The function table describes the input-output relationships. An example of both the functional and operational state tables can be seen in Tables 3.3 and 3.4. Figure 3.1 shows the visual relationship of the tables. Majdara and Wakabayashi introduced a new algorithm to generate a fault tree based on an occurrence of an undesirable event being defined. The algorithm uses the input and output connections between the components of a system in order to trace the cause of the undesirable event. The algorithm traces back from the occurrence and identifying component states or outputs that could cause the event. These would then be used to generate the fault tree.

When constructed the fault tree is checked for consistency. Where consistency means that two mutually exclusive events cannot occur at the same time. The algorithm would then remove these.
Table 3.3: Operator-driven valve state transition table

<table>
<thead>
<tr>
<th>First State</th>
<th>Command</th>
<th>Functionality</th>
<th>Condition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>2</td>
<td>OK</td>
<td>Close</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>2</td>
<td>Fail-to-close</td>
<td>Open</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>1</td>
<td>–</td>
<td>Open</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>1</td>
<td>OK</td>
<td>Open</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>1</td>
<td>Fail-to-open</td>
<td>Close</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>2</td>
<td>–</td>
<td>Close</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>0</td>
<td>–</td>
<td>Open</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>0</td>
<td>–</td>
<td>Close</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Operator-driven valve function table

<table>
<thead>
<tr>
<th>Flow input</th>
<th>State</th>
<th>Flow Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>–</td>
<td>Close</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Open</td>
<td>1</td>
</tr>
</tbody>
</table>
3.2.2 Digraph Method

Another method for constructing fault trees is presented by Lapp and Powers (1977) by using digraphs or directed graphs. This method details how taking a constructed digraph can be transformed into a fault tree.

A digraph consists of two components; nodes and directed edges. These nodes are used to represent process variables, and also some types of failures. Relationships between these nodes are through edges that connect between them. If a deviation occurs in one variable and has an effect on another variable in the system, then there is an edge connecting these variables. These deviations can either be positive or negative; “+” and “-” respectively. Depending on the severity of these deviations the values 0, 1 and 10 are used to represent none, moderate and a very large deviation, respectively. For example in a system that deals with mass flow rates, should there be a large decrease in that flow then there would be a (-10) associated with that connection.

The digraph is used to detect control loops to identify ‘disturbances’ which could result in the top event. By following the disturbances through the digraph the fault tree can be constructed by allowing the disturbances to propagate.

3.2.3 Modified Decision Table method

A new method of combining both decision tables and digraphs to automatically construct fault trees was introduced by Henry and Andrews (1997). Decision tables brought forward the advantages of the ability to identify the normal state of a system and digraphs the ability to detect and classify control loops.

This method is demonstrated in a program that uses AutoCAD schematic diagrams and decision tables to model the system and its components. The AutoCAD diagram is used as a method of showing how the components of a system link to one another. This allows this sort of analysis to be completed during the design phase of a new or modified system. The decision tables are used as a method of modelling the components and are stored within a library of decision tables. These decision tables can be edited by the user to suit their needs using a component editor. To generate the fault tree a top event is entered as input from the user into the program. The program then traces the undesirable event and generates a fault tree based on the trace. This program does not complete the analysis of the fault tree, but does provide a file version of the fault tree that can be read by commercial packages.

The method for automating the construction requires the system description, which is given in the form of an AutoCAD schematic diagram. From the schematic diagram a file is created that stores the topology information for the system. This information includes how the components connect together in the system. A library of component decision tables
exists and these can be amended or new decision tables generated. This is accomplished using a generic component editor. A top event is entered into the fault tree construction program so that the causes of such an event can be traced. The program then generates the fault tree structure for the top event. This fault tree structure is written to an output file, which is capable of being read by other commercial analysis packages. These packages can then be used to carry out the qualitative and quantitative analysis of the fault tree and therefore obtain the top event probabilitys. As this method makes use of a Computer Aided Design (CAD) interface, this can allow the reliability assessment of a system at the design stage.

Henry and Andrews created a modified decision table method to overcome the main weakness of decision tables: the inability to detect, classify and analyse control loops and circuits. This was overcome by modifying the tables to incorporate a gain that is used to show the connection between the inputs and the outputs of a component. An example of a modified decision table is given in Table 3.6.

This new decision table element, to add gains, allows for the identification of circuits within a system. By using the component decision tables and the system topology Henry (1996) generates a digraph. The nodes of the digraphs represent the outputs of a component and the edges link these nodes together. By defining a top event and using the digraph a new item is created, a topology graph. The fault is traced and the nodes that are ‘passed’
are identified. This process identifies any circuits within the system.

### 3.2.4 Cause-Consequence Diagrams

Cause-Consequence Diagrams (CCDs) are another method of showing the failure logic of a system in a similar fashion to fault trees. However, CCDs identify a complete set, rather than a sub-set, of consequences as a result of an initiating event. CCDs can become very large particularly if applied to industrial scale systems and is a subjective process. To remove human error from the construction of such diagrams Valaityte et al. (2010) discusses a method of automatically constructing a CCDs based on a system description in the form of a system topology diagram and decision tables describing component behaviour. The construction algorithm detailed within Valaityte et al. (2010) is based on whether the system contains electrical circuits. This changes the way in which the system is handled. Circuits are identified in this method by locating loops using the system topology diagram. This method uses the initiating event to follow/trace the path of the knock-on effects within the system and identifying these within the CCD.

### 3.2.5 Mini fault trees

Taylor (1982) developed an algorithm for fault tree construction using mini fault trees. The initial information given was the description of the system, which describes each component and the set of connections between these components. Each component type stores in a library a standard function and a failure model. These functions and models are made up of mini fault trees. The mini fault trees consist of the following:

- **Input event**
- **Set of component conditions**
- **Set of output and state change events**

This is a method of describing the component’s behaviour in fault tree form rather than decision tables as seen in previous methods. This method uses the undesirable event as the starting point and identifies the event and the associated component. The method then moves through the connections of the component to other components in the system that also have the event listed. The process continues to follow associated components and events and further following new events as a result of a previous event. Using the associations the fault tree is constructed.

### 3.2.6 Faultfinder

The FAULTFINDER program was developed at Loughborough University in the Department of Chemical Engineering and written in Fortran 77. The programme has been
described in detail in 4 papers by Kelly and Lees ((1986a) – (1986d)). These papers are split into the following four categories; the modelling method (Kelly & Lees 1986a), the fault tree synthesis method (Kelly & Lees 1986b), the interactive facility by implementing these (Kelly & Lees 1986c) and illustrative examples (Kelly & Lees 1986d).

The third paper (Kelly & Lees 1986c) gives a detailed description of the main subprograms that comprise of the suite of programs within FAULTFINDER. These are listed below:

1. Master Program (MASTER)
2. Unit Model and Event Model Programs (MODGEN and EVTGEN)
3. Fault Tree Generation Program (FLTGEN)
4. Fault Tree Analysis Program (FLTANL)
5. Fault Tree Display and Evaluation Program

An example of the structure of the FAULTFINDER programme is given in Figure 3.2. The sub-programs of FAULTFINDER are described in detail below; the uses of each sub-program and their relevance to the other sub-programs are considered.

3.2.6.1 Master Program

The master program or MASTER is the framework program which handles the data inputs and outputs. In MASTER there are two options; the first is to call a fault tree synthesis
program and the second is to transfer input data straight into a process control computer. For the first option a top event is selected and a fault tree is generated using FLTGEN. This creates a fault tree encoded as an array. Using other facilities within FAULTFINDER the fault tree can be viewed, modified, plotted (using GINO graph plotting facility) and analysed using FLTANL. The second option, at the time of the publication of the paper, was not developed, but was available in principle. The structure of the control program can be split into six parts:

- Input the configuration data.
- Input unit model data from MODLIB and EVTLIB.
- Attachment of FLTGEN: In this part the options for the fault tree generation and drawing are selected. The top event data is also input here.
- Generation of the cut sets.
- Configuration editor for modifying the configuration.
- Input of the sequence abort conditions. The configuration is also modified at this part for the next stage sequence.

3.2.6.2 Unit Model and Event Model Programs

A unit model contains the necessary data for a particular component within a system. The collection of unit models should make up the system as a whole. These unit models are created in MODGEN and then stored in the unit model library, MODLIB. The unit model program, MODGEN, is an interactive program where the necessary data for each unit model is entered.

The model data that is required for each has a specified format. The information first given is the model name which can be an alphanumeric description and the model number which lies between 1-100. The information required next is the engineering description which defines the equipment and engineering assumptions (these relate primarily to the propagation equations explained shortly). These are both given in plain text format. The next group of information is the system information including propagation equations, event statements, and decision tables if applicable to the system. This information provides the initial behaviour of the unit, each is described below:

- **Propagation Equations**: Propagation equations describe how a fault moves through units in the system by describing the relationship between an output variable of a unit and the input and other output variables of the unit.

- **Event Statements**: There are three types of event statements that are considered;
- Initial event statements: These statements are used to represent the affect of a fault initiating a sequence of disturbances or an operator action as an initiating event. These statements are also used in place of a propagation equation when variable deviation cannot be derived.

- Intermediate event statements: These statements are used to include logical relations such as AND gates and can also be used to represent generic faults. Intermediate events can have either faults or variable deviations as their causes.

- Terminal event statements: These statements are used to represent a variable deviation that can cause a terminal fault. This approach is not always used within a unit model as this can cause the size of the model to increase with a full list of terminal events. Therefore these are usually given in a separate event model.

- **Decision Tables:** Decision tables can be used instead of initial event statements when the logical relations become complex. For example, equation 3.2.1 can be represented as Table 3.7, where T represents True and – represents Don’t care.

\[
Z = (A \text{ AND } B) \text{ OR } (C \text{ AND } D) \tag{3.2.1}
\]

Information about the components normal, failure states/modes is also included in the system information. The format for the output data is in the form of mini-trees; a list of mini-tree top events and a list of the individual mini-trees.

Event statements and decision tables are the most flexible out of the three input data and therefore widely used. However if decision tables are the primary model, then they are generated manually. This takes time and storage on the system. Therefore the use of decision tables was kept for complex models.

The method chosen for the output information was in the form of mini-fault trees. These are well adapted for the automatic generation of models and fault trees. Mini-fault trees can be created manually, but it is more desirable to automatically create them. A unit model should only be created when there is a need for one. This ensures that the model has been tested before it is stored to the library.
3.2. Methods for Automation of Reliability Models

Mini-fault trees are generated from the propagation equations and the event statements or decision tables. The propagation equations can be converted into the mini-fault trees, as top event, or output is variable deviation given on the left hand side of the equation, with the basic events, or inputs to the tree are the variable deviations on the right hand side of the equation. Mini-fault trees generated from these equations can be incomplete and can require information from initial event statements and decision tables. If only a propagation equation is required to form the mini fault tree then an OR gate is used, but if an initial event statement or a decision table is required then an AND gate is required.

In some cases it is more useful to model as an event rather than a unit. The types of events that this would be applicable to are:

1. Undesired events
2. Physical and phase changes
3. Materials

Undesired events are those events that are terminal events, as discussed earlier. These can either be modelled with unit models, but are more often modelled with event models due to the problems of the size of the model with unit models. Event models are generated by inputting the data into the event model program, EVTGEN. The event model is then automatically stored in the event model library, EVTLIB.

The input information for EVTGEN is similar to that for the input information for MODLIB, the first piece of information to create an event model is the event type, such as those mentioned above; undesirable event, phase or physical change, or materials failure. Event name and number, where the name is an alphanumeric description and event number that is in the range of 1-20. Then a description of the event; event statements and decision tables (when applicable). The output from the program is mini-fault trees.

3.2.6.3 Fault Tree Generation Program

The fault tree generation program, FLTGEN, is a sub-program designed for fault tree generation, plotting and manipulation. The program generates a fault tree as an array of data before creating a drawing of the tree. To create the data, an undesirable event is specified as the top event, and from there information of how the system would reach such an event is compiled. To do this individual branches would be created from the appropriate mini-trees. Starting with the top event, this would be the top event of a mini-fault tree within an event model. This tree would be placed directly below the top event and the input events would be mini-top events of other mini-fault trees that are either within the same event model or a unit model connected to it. This creates the next level of mini-trees. This process is continued until only basic or diamond events are left.
Consistency checking is completed on the final tree to check that none of the boundary conditions have been broken and faults that are not allowed to occur, haven’t. Series and parallel consistency checks are required. Series consistency check each branch individually for event consistency and this can be done whilst the tree is generated. Whereas parallel consistency checks all branches against the others under an AND gate for event consistency, but can only be done once the synthesis is complete.

Due to the number of unit models used to generate a fault tree a number of minor or repeated faults can exist. The fault tree can also contain looped events or variable deviations that exist elsewhere in the tree. These are removed or suppressed to unclutter the tree with unnecessary information. When a looped event is identified this is suppressed and a diamond event is put in its place. By doing this the stored fault tree is affected as the looped events are not created, but instead point to the one and only occurrence in the tree. When a repeated fault is identified then these were removed by either full or partial suppression. Full suppression removes the types of basic events from the stored and drawn tree, whereas partial suppression only removes them from the drawn tree and not the stored tree.

The fault tree is plotted and can be generated on screen. The tree is plotted using GINO graph plotting facility. GINO generates the tree; producing its structure and a graphical output. The tree generation has two parts; the first is the initial synthesis and the second is the rationalisation process. This yields the fault tree.

### 3.2.6.4 Fault Tree Analysis Program

The sub-program fault tree analysis, FLTANL, has only two features; the first is executed in MASTER, where the cut sets of the tree can be determined. The second is with the use of other software packages specifically designed for fault tree analysis. One such package is PREP and KITT (Vesely & Narum 1970). This is separate from the MASTER program. This consists of three programs:

- **TRIAD**: Analyses and rearranges
- **CUTSET**: Calculates minimal cut sets
- **KITT**: Calculates top event frequency

The first two are part of the PREP, or preparation of the data before the top event frequency can be found. Then KITT calculates the top event frequency.

### 3.2.6.5 Fault Tree Display and Evaluation Program

As mentioned earlier the tree produced in FLTGEN can be plotted and manipulated. The Fault Tree Display and Evaluation Program extends this manipulation to editing the tree
itself in terms of its structure to create a clearer structure and the descriptive text of the events to a comprehensible form.

The editor can also be used for general fault tree creation which can be useful for simple structures.

3.3 Summary

The methods presented here were very focused on the automatic generation of fault trees. None of the methods researched constructed automatically a system undergoing a phased mission, which implies that there is room for the work presented here. One of the methods discussed in this section by Majdara and Wakabayashi (2009) was a method to trace a fault using decision tables. This method provided a starting point as to how the components of systems in this work would be described in order to generate a reliability model.
Chapter 4

Modelling of Non-Repairable Systems

Contents

4.1 Introduction .................................................. 87
4.2 Model Inputs .................................................. 88
  4.2.1 Component Description .................................. 88
  4.2.2 System Description ..................................... 91
  4.2.3 Circuit Description ..................................... 92
  4.2.4 Phase Description ....................................... 92
  4.2.5 Initial Conditions ....................................... 92
4.3 Petri Net Models ............................................. 93
  4.3.1 Component Petri Nets ................................... 93
  4.3.2 Circuit Petri Nets ...................................... 102
  4.3.3 System Petri Nets ...................................... 104
  4.3.4 Phase Petri Nets ...................................... 109
4.4 Algorithm .................................................... 115
4.5 Summary ..................................................... 120

4.1 Introduction

By automating the process of building the reliability model, the process of assessing a system’s reliability becomes more efficient and reliable. To achieve this, the reliability model generated must be versatile and easy to simulate once constructed. For these reasons, Petri nets have been chosen as the modelling method. To construct the Petri nets, the system and mission that the system is undertaking must be described sufficiently. To achieve this, two techniques have been employed to describe the components within a system; decision tables as discussed in Chapter 3 and the method by Majdara and Wakabayashi (2009).
4.2 Model Inputs

The information required from the user includes a description of each component within a given system, which includes the component’s failure and, when applicable, repair rates, and the number of input and output connections the component has. This information can then be used to create a system map that shows how each of the component’s connections link to other components in the system. The mission of the system is also required from the user; this details how a system can succeed or fail within each stage, or phase, of the mission.

4.2.1 Component Description

Each component in the system is described using a decision table. If a component has more than one operational mode, another table is also required: the operational mode table, which describes how each mode affects the component’s behaviour. Operational Mode tables are similar to state transition tables introduced by Majdara and Wakabayashi (2009) and described in Section 3.2.1. A decision table is still required alongside the operational mode table as the operational mode table describes the internal conditions of a component, whereas the decision table is used to describe how the component interacts within the system scope. An example of a component with a single operating mode is a power supply, shown in Figure 4.1. The decision table for the power supply is given in Table 4.1, where $C$ and $NC$ represent current and no current, respectively.

For components with a single operational mode, the headings in the decision table are $in$, $state$ and $out$. These correspond to the inputs and outputs to and from the component and the state of the component, where the state is the working, failed or repair state of the component. Depending on the system structure there may be multiple inputs and outputs, as shown in Section 3.2.1. If a component is time dependent, then a $time$ column
is necessary to capture this information. This can be particularly relevant to the time of a phase within a given mission.

For components that have more than one operating mode, both tables are necessary. A link is therefore required between the two tables. This is used to show how the internal mode of the component affects the system. To show this connection the decision table has a heading \textit{mode} in place of the \textit{state} heading in a normal decision table. The state of the component is dealt with within the operational mode table. An example of a component that requires an operational mode table and a decision table is a toggle switch. A toggle switch requires an input from another component (usually an operator) and the input can change the operating mode of the toggle switch. The decision table and the operational mode table for the toggle switch can be seen in Table 4.2 and Table 4.3, respectively. Where the command values \textit{CL}, \textit{OP} and \textit{NA} represent closed, open and no action, respectively and the state values \textit{W}, \textit{FCL} and \textit{FOP} represent working, failed closed and failed open, respectively. The table headings relate to the inputs and output shown in Figure 4.2.

\begin{table}[h]
\centering
\begin{tabular}{lll}
\hline
In 2 & Mode & Out \\
\hline
1 & – & Open NC \\
2 & NC & – NC \\
3 & C & Closed C \\
\hline
\end{tabular}
\caption{Decision Table for toggle switch}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{llll}
\hline
Mode 1 & Command (In1) & State & Mode 2 \\
\hline
1 & Closed & – & FCL & Closed \\
2 & Closed & CL & – & Closed \\
3 & Closed & OP & W & Open \\
4 & Closed & NA & – & Closed \\
5 & Open & – & FOP & Open \\
6 & Open & OP & – & Open \\
7 & Open & CL & W & Closed \\
8 & Open & NA & – & Open \\
\hline
\end{tabular}
\caption{Operating mode table for toggle switch}
\end{table}
An operational mode table has four headings. Two relate to the mode of the component: *Mode 1* relates to the initial mode, or starting mode and *Mode 2* relates to the resultant mode. The other headings within the operational mode table are *command* and *state*, where *command* relates to the input that can cause a change in the mode of the component. *State* is the same as the state discussed in decision tables, representing the component’s working or failed state. *Mode 2* in the operational mode table is the mode value related to the decision table heading *Mode*.

The principle idea is for the software to contain a list of common component decision and, where applicable, operating mode tables. The user would then only be required to create component tables for those components that are not commonly used. A number of component tables have been generated for the example systems seen throughout the next few chapters and are re-useable for other users.

The above information describes how the component behaves, but does not describe how it fails. This is covered by the failure rate of the component, which can be constant or follow a distribution. This information must also come from the user as it is not within the scope of the software to hold failure rates of components, which can be variable depending on manufacturer and version. The repair rate of a component will be covered in later chapters.

### 4.2.1.1 Time Dependencies

A component’s behaviour within a system can sometimes be dictated by time, i.e. during different time periods the component can behave in different ways. These times would relate to the phases of the mission undertaken by the system, in which this component resides. A good example of a component that acts in this way is a timer relay. A timer relay initially energises when current is applied and, once a specified time has elapsed the relay de-energises. This type of information is required in the decision table of the component, to define what period of time, or specific time, the component changes, and how that change affects the component’s outputs. The timer relay decision table is given in Table 4.4 and this shows that the timer relay has one input of an electrical connection and two outputs. The first output is to another component, such as a contact (*EN* and *DE*, energise and de-energise, respectively), and the second output is another electrical connection. In the table another column heading is included, *t*, representing time. The rows that include a time element can only occur at the time or time frame stated. In the example the first row can only occur between the time zero (inclusive) and some predefined time, *t_1*, but not at *t_1*. This row states that the outputs would be *EN* and *C* during this time period, as long as the pre-requisites from the *input* and *state* columns are met. The second row states that this row can occur at time *t_1* and any time greater for a given mission. The outputs for the row are different from the previous row, with *DE* and *NC*. Although both
4.2. Model Inputs

Figure 4.3: Example of a simple system with one circuit

![Simple System Diagram](image1.png)

Figure 4.4: Example of a topology diagram

![Topology Diagram](image2.png)

Rows have the same input and component state they produce different outputs depending on the time/time frame stated.

<table>
<thead>
<tr>
<th>Table 4.4: Timer Relay Decision Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

4.2.2 System Description

The system description describes how the components link together. This information is presented in the form of a system topology diagram. This information must come from the user, but the labels of $IN$ and $OUT$ may be pre-defined if the user uses a library component decision and operational mode tables. An example of a simple system is given in Figure 4.3 and the topology diagram for this system is given in Figure 4.4. This system has three components with unique identifiers, $PS$, $M$ and $FS$, which represent a power supply, motor and fuse, respectively. If there were more than one component of the same type, then the identifiers would be numbered, for example $PS1$, $PS2$ for multiple power supplies. Within the system topology diagram the outputs of each component, $OUT$ are linked to the input, $IN$, of the next component in the system structure. These $IN$ and $OUT$ labels map to the decision and operational mode table $In$ and $Out$ columns.
4.2.3 Circuit Description

Circuit descriptions are only required when the system contains electrical components. The circuit description is in the form of a list containing the unique identities of components in an electrical system. Circuits within a system need to be identified so as to determine whether or not current flows in the system. This is completed by using the system and component information described earlier in Sections 4.2.2 and 4.2.1, respectively.

The circuit models contain the critical states or modes of the components within the circuit. If a component has more than one operating mode, then the mode can change the circuit’s state. If a component does not have more than one operating mode then the state of the component, working or failed, will affect the circuit. For the creation of the circuit model the components within the circuit need to be identified along with the mode or state that can create current or no current in the circuit. The decision tables of the components hold this information and the software identifies this automatically.

It should be noted that within the software there is a location for the pre-determined values used for $C$, $NC$ (current and no current), which is used as part of the circuit list generation. These values can be altered if necessary by the user.

4.2.4 Phase Description

The phase description specifies the mission that the system is to undertake. This information will be given in the form of a table: the phase transition table. The phase transition table describes the phases that the system undertakes, assuming the system works from the beginning to the end of the mission. This details the length of the phase and the condition that causes the phase to transition to the next phase. As the system may not always work from the beginning to the end of a mission, other phases are required to show system failure. The number of these phases depends on the system. All of this information is included in the phase transition table.

It should be noted that each row of the phase transition table does not always have time associated with it. In some cases a component’s output can be the trigger of a phase transition.

The model generated from the phase transition table provides a method of monitoring the time during a mission and will be covered in a later section.

4.2.5 Initial Conditions

The initial conditions required before a simulation can begin include the following information:

- The starting mode of any components with more than one operating mode.
• The initiating component that starts the system, and the input/output condition of the component is required. For example a switch closing could be the initiating condition, therefore it would be the switch with an input to close, $CL$.

• It is assumed that all components are in a working condition at the beginning of a simulation.

4.3 Petri Net Models

In this section the Petri models, CPNs, SPN, Circuit Petri Net (CiPN) and PPN are described. In the previous section the input information required to create each of these models was discussed. This section shows how the input information is translated into these models by the software.

4.3.1 Component Petri Nets

The component description given in Section 4.2.1 is now used to generate the CPN. The decision and operating mode tables described in section 3.2.1 are used in the construction of the Petri net for the component.

4.3.1.1 Component Petri Net Construction

To create the CPN, each row of the decision and operational mode tables is taken as a single transition. For decision tables, the values from the input and state/mode columns form the input places to the transition and the output column values form the output places.

For operational mode transition tables, the values from the starting mode, command and state columns form the input places to the transition and the resulting mode column values form the output places.

If there is a time column associated with a decision table, this will be dealt with through the PPN to be discussed later.

Construction Procedure

The procedure for constructing the CPNs is as follows:

1. Except for the time, $t$, column, identify types in each column, e.g. $C$ and $NC$. Ignore “–” entries.

2. Take each column and

   (a) If column is $mode1$, $mode2$ or $mode$, then create one place for each type that exists in all of these columns.
(b) Else, create a place for each type.

3. Create an immediate transition for each row of the table.

4. Input places for a transition for an OMT table are mode1, in and state. Input places for a DT are in and either state or mode. Taking each row of the table at a time, create an arc from the row’s input place to the transition for that row (created in step 3) as follows:

   (a) If in column type, create single headed arc.
   (b) If state column type, create double headed arc.
   (c) If mode column type, create double headed arc.
   (d) If mode1 column type, create single headed arc.

5. An output place(s) from a transition for an OMT table is mode2 and for a DT is out. Taking each row of the table at a time. Create an arc from the transition to the place representing column type depending on the column:

   (a) If out column type:
      i. If the only input arc to the transition is a double headed arc then a single headed inhibit arc is used.
      ii. Else, use a single headed arc.
   (b) If mode2 column type:
      i. If the only input arc to the transition is a double headed arc then a single headed inhibit arc is used.
      ii. Else, use a single headed arc.

4.3.1.2 Decision Table Example

To demonstrate the procedure, an example is shown in Figure 4.5 for a power supply. For step one, Figure 4.5a, the types in the table are identified; \{C, NC\} in column in, \{W, F\} in column state and \{C, NC\} in column out. Step two moves through each column, creating a place for each type identified in step one, as seen in Figure 4.5b. For step three, the number of rows in the table is counted and an immediate transition is created for each row (three in this case). This can be seen in Figure 4.5c. Step four moves through each row of the table and generates arcs between the types in the in/state columns and the transition representing that row. For example, on row one of the table, the in column type C and the state column type W are the inputs to the first transition. The type of arc is dependent on the column heading. For in column types a single-headed arc is required. State column types require a double headed arc as the component can only change state.
when the time to failure is reached and is not dependent on whether current flows or not through the component. This is demonstrated in Figure 4.5d. The fifth and final step is to create the arcs from the transitions to the output type places. The type of arc required is dependent upon the arcs that are connected to the transition from step four. If the only input arc connected to the transition is a double headed arc then an inhibit, single headed arc is required. This stops the possibility of continuous firing of tokens from the state places. Otherwise a single headed arc is required from the transition to the output place. An example of each type of arc can be seen in Figure 4.5e. This is the final part of the Petri net to be added at this stage. The next to consider is the component failure times, discussed in the next section.

To demonstrate the procedure for a component with multiple operating modes, the toggle switch is used. Applying the steps to this component can be seen in Figure 4.6 and Figure 4.7. Figure 4.6 is the construction process for the operational mode table and Figure 4.7 is the construction process for the decision table. It is not necessary for the operational mode table to be assessed first, but for the purposes of demonstrating the procedure this was considered first.

**4.3.1.3 Operational Mode Transition Table Example**

Starting with the operational mode table the first step is to identify the types in each of the columns of the table. This can be seen in Figure 4.6a. The types identified in the *mode* 1 column and the *mode* 2 column should be identical. The next step in the procedure creates a place for each type under all columns except for the *mode* 1 and *mode* 2 columns. Places for the types in these columns are created once for both columns. The purpose of the places for modes is to show the current mode of the component; this cannot be done if there are multiple places representing the same information. This can be seen in Figure 4.6b. The next step is to create a transition for each row of the table, as seen in Figure 4.6c. In step four the arcs used as input to the transition are generated, depending on the input place and the row of the table the transition relates to. If the place is from the *mode* 1 or *in* 1 column then a single-headed arc is used. If the place is from the *state* column then a double-headed arc is used. The final step of the procedure is to create the output arcs from the transition. As the only output is *mode* 2, the arcs are single-headed. The final Petri net from the operating mode table of the toggle switch can be seen in Figure 4.6e.

As the toggle switch has multiple operating modes the procedure is applied to the second table, the decision table. To begin step one is completed by identifying the types in the columns of the decision table, as seen in Figure 4.7a. As the types identified in the *mode* column were already identified during the process for the operational mode table, these types do not need to be generated as places in step 2. The remaining columns, however, do need to have a place created for each type, this can be seen in Figure 4.7b.
Figure 4.5: Procedure steps for the construction of a Petri net representing a power supply
4.3. Petri Net Models

![Diagram showing the construction of a Petri net model for a toggle switch using the OMT](image)

Figure 4.6: Procedure steps for the construction of a Petri net representing a toggle switch using the OMT.
For step three the number of rows in the table is counted and an immediate transition is created for each row, as shown in Figure 4.7c. Step four moves through each row of the decision table and creates an arc between the place representing in 2 column types and the mode column types. Arcs are added according to the column types, as before. This can be seen in Figure 4.7d. The last stage of the procedure creates arcs from each of the transitions. These arcs go from the transition of the row to the out column type associated with the row. As before if the only input into the transition is a double-headed arc then the output arc is a inhibit single headed arc. The result of the procedure applied to the decision table and the operating mode table is seen in Figure 4.7e.

### 4.3.1.4 Component Failure

To account for the failure of a component, transitions are required between the working and each possible failure state. For components with only one mode and one failure state, there is only one working and one failed state, therefore a single transition is required between the working and failed state. This transition is dictated by the failure rate or failure distribution of the component, and therefore the transition is a delayed transition where the delay is the time to failure. An example of how this is represented in Petri net form is given in Figure 4.8a, where \( W \) represents the working state and \( F \) represents the failed state of the component.

For components with more than one mode, this becomes slightly more complex. The mode the component currently exists in must be accounted for in order to show which failure mode has occurred. To demonstrate this an example is given in Figure 4.8b of a component with two operating modes, \( M1 \) and \( M2 \), where both modes have the same time to failure, \( t_F \). Whether multiple failure modes for a component have the same failure time depends on the component and the operational mode. This information will need to be provided as the input to the model. Another example of this can be seen in Figure 4.8c; this figure shows that sometimes the component can have a different failure rate for each time of failure. In this figure there are two failure modes, \( FM1 \) and \( FM2 \), and each failure mode has a different time to failure, \( t_{FM1} \) and \( t_{FM2} \). The Petri net works in the same way as Figure 4.8b.

There are also components that have multiple failure states but do not have multiple operational modes. An example of such a component is a pressure gauge. If the main failure of a pressure gauge is to fail stuck, then there is a possibility of the pressure gauge failing at different levels on the gauge. Depending on the system, the values on the gauge could be classified into different groups, for example, \( LOW \), \( HIGH \) and \( VHIGH \) (low, high and very high pressure). To transition from the working state to one of these failure states the system must first transition to a dedicated failure place. This requires the normal transition as seen in Figure 4.8a. To transition to the relevant failure state the information
4.3. Petri Net Models

<table>
<thead>
<tr>
<th>In</th>
<th>Mode</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>closed</td>
<td>C</td>
</tr>
<tr>
<td>-</td>
<td>open</td>
<td>NC</td>
</tr>
<tr>
<td>NC</td>
<td>-</td>
<td>NC</td>
</tr>
</tbody>
</table>

(a) Identifying the types in each of the columns

(b) Creating a place for each column type

(c) Creating a transition for each row of the table

(d) Creating an arc from each input place for each transition

(e) Creating an arc from each transition to the output places

Figure 4.7: Procedure steps for the construction of a Petri net representing a toggle switch using the DT
from the component’s decision table is used. Within the decision table the different failure states should be listed with their appropriate outcome in the output, \( OUT \), column. The decision table should have full coverage of the mission so the failure state can be determined at any point during the mission. The failure state can then be determined using the other rows of the decision table. The other rows of the table have an output that relates to one of the outputs linked to a failure state. By using this information a transition to represent each row of the decision table not related to a failure state is generated to show the transitions into the failure states. An example of a pressure gauge decision table can be seen in Table 4.5, where \( t_1 \) is a period time in which the input, \( In \) can change, and the corresponding work to failure state transitions can be seen in Figure 4.9. The construction procedure to generate the transitions is described below:

1. This component has been identified as having multiple failure states and does not have multiple operational modes. Create a place to represent that the component has failed.

2. Create a timed transition and set the time to failure as the value generated using the distribution identified for this component.

3. Create a single headed arc between the working place of the component and the timed transition created in 2.

4. Create a single headed arc between the timed transition and the failed place created in 1.

5. Moving through each row of the table: If the next row in the table is not associated with a failure state then complete the following:
   (a) Get the output types associated with this row of the table.
   (b) Create an immediate transition.
   (c) Create a single headed arc between the failed place created in 1. and the immediate transition created in 5b.
   (d) Using the output types identified in 5a identify the failure state would cause the same output types.
   (e) Create a single headed arc between the immediate transition and the failure state identified in 5d.
   (f) Get the input column types from the row and create a double-headed arc between the place representing the input type and the immediate transition.

This representation of the component’s state is incorporated into the CPNs by linking to the places that represent the working and failed states of the component.
4.3. Petri Net Models

(a) Example Petri net of the failure rate for a single mode component

(b) Example Petri net of the failure rates for a multiple mode component where the failure rates are the same

(c) Example Petri net of the failure rates for a multiple mode component where the failure rates are different

Figure 4.8: Example of using failure rates in Petri net models

Figure 4.9: Example of a component with multiple failure states and one operating mode
Table 4.5: Decision table for a pressure gauge

<table>
<thead>
<tr>
<th>t</th>
<th>In</th>
<th>State</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t &lt; t_1$</td>
<td>CONST</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>$t &lt; t_1$</td>
<td>INC</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>$t_1$</td>
<td>CONST</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>$t_1$</td>
<td>INC</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>DEC</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>–</td>
<td>F_LOW</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>–</td>
<td>F_HIGH</td>
</tr>
</tbody>
</table>

4.3.2 Circuit Petri Nets

4.3.2.1 Circuit Petri Net Construction

CiPNs are necessary to track the flow of current in electrical systems or subsystems. A CiPN is required for every electrical circuit identified in a given system. Once the components have been identified in a given circuit the decision tables of those components can be searched. Moving through each row of the decision tables, the component states/modes that can cause current/no current within a circuit are identified.

From this information the Petri nets for ‘current in circuit $n$’ and ‘no current in circuit $n$’ are developed. For the Petri net ‘current in circuit $n$’, all components in circuit $n$ need to pass current. For ‘no current in circuit $n$’, it only takes one component in circuit $n$ to not pass current.

Construction Procedure

1. Take circuit list $n$ and create a place representing “Current in circuit $n$” and a place representing “No Current in circuit $n$”.

2. For “Current in circuit $n$” and “No Current in circuit $n$” identify the rows in the decision tables of the components in circuit list $n$ that have an out column that represents this circuit having current and no current.

3. For each row identified:
   
   (a) If “–” exists in either state or mode column then ignore that row.
   
   (b) Else, identify type in state or mode column.

4. For “Current in circuit $n$”, create a single immediate transition and for “No Current in circuit $n$” create an immediate transition for each row identified in step 2.

5. Create a single headed inhibit arc from the transition(s) to the places representing “Current in circuit $n$” and “No Current in circuit $n$”.
6. For each state or mode type identified for “Current in circuit \( n \)”, create a double headed arc from the place representing this state or mode in the CPN to the transition.

7. For each state or mode type identified for “No Current in circuit \( n \)” create a double headed arc from the place representing this state or mode in the CPN to the transition representing that row.

Each CiPN connects to a component within the circuit list through the transitions created from the decision tables. The component that is identified is determined by the software by using the system topology information and the circuit list.

To demonstrate the procedure the example given in Figure 4.3 is extended to include another circuit as shown in Figure 4.10. In this new system there are two power supplies, \( PS_1 \) and \( PS_2 \), a fuse \( F \) a motor, \( M \) and two junctions, \( J_1 \) and \( J_2 \) as labelled in the diagram. From the diagram there are two circuits that exist in this system and are listed below:

1. \{ \( PS_1, J_1, FS, M, J_2 \) \}

2. \{ \( PS_2, J_1, FS, M, J_2 \) \}

The procedure begins with step one using the circuit lists given above. Taking circuit 1 first, a place is created to represent ‘current in circuit 1’ and another to represent ‘no current in circuit 1’. Each row of each component decision table is considered, to find any rows that result with an output (\( out \)) with \( C \) or \( NC \). As seen in Figure 4.11b, all rows of each of the tables have a \( C \) or \( NC \) output and therefore all must be considered for the next step. Step three moves through each of the rows identified in step two and identifies which state (or mode when applicable) of the component leads to either current or no current. Figure 4.11c shows the states identified for each of the three components. Figure 4.11d shows the Petri net representation of the information found in this step. These
places representing the components’ states are linked to the places within the individual CPNs, similar to those found in Section 4.3.1.1. The next step, step four, generates a single transition which links to the ‘current in circuit 1’ place. Step four also generates a transition for every component state that can contribute to ‘no current in circuit 1’. Figure 4.11e shows these transitions. Step five adds the arcs between the transition and the places for ‘current in circuit 1’ and ‘no current in circuit 1’. A single, single-headed inhibit arc is required between the transition and the places. An inhibit arc is required to ensure that firing of the transition will not occur constantly. Steps 6 and 7 generate double-headed arcs from the component state places and the transition. All arcs from the places representing component states that lead to current connect to the same transition, as seen in Figure 4.11g. Each component state that leads to no current have an individual arc to connect to, as seen in Figure 4.11h.

This same procedure was also applied to the circuit 2 list producing the Petri net in Figure 4.12.

If a component with multiple operational modes was included in the circuit, such as the toggle switch then the method would be the same, except instead of the state identified in step three it would be the component’s mode.

The CiPNs indicate the current state of the circuit at a given time, therefore a Petri net extension is required to track the changes within the circuit. An example of this extension can be found in Figure 4.13. Another purpose of this extension is to ensure that there is not a constant flow of tokens from the CiPN. The first iteration of the connection of the CiPN to the SPN proved that a simple link between them would cause a significant increase in the number of tokens moving around the SPN. This would cause multiple tokens to exist in a single place at any given time. The method demonstrated here reduced the likelihood of too many tokens moving through the SPN.

Where the connection between the CiPN and the SPN occurs is an automated decision made by the software. From the initiating component a flow of connections is explored. When the first component in the circuit is found, this becomes the connecting component. The software establishes the connection between the CiPN and the SPN on this component.

4.3.3 System Petri Nets

4.3.3.1 System Petri Net Construction

The SPN is formed from the individual CPNs (described in Section 4.2.1) to create a single model of the overall system. The CPNs are connected together according to the system topology.
4.3. Petri Net Models

(a) Create places representing the current and no current for circuit 1

Decision Table for PS1 | Decision Table for FS | Decision Table for M
<table>
<thead>
<tr>
<th>In</th>
<th>State</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>W</td>
<td>C</td>
</tr>
<tr>
<td>-</td>
<td>F</td>
<td>NC</td>
</tr>
<tr>
<td>NC</td>
<td>-</td>
<td>NC</td>
</tr>
</tbody>
</table>

(b) Moving through the component decision tables and identifying the rows containing an output with $C$ and $NC$

(c) Within the rows identified in the decision table, identify the state/mode of the component that can cause the resultant output of $C$ or $NC$

(d) Diagram representation of the states identified which have an output of $C$ and $NC$

(e) Transitions generated for both conditions, one for current in circuit 1 and one per component state identified for no current in circuit 1

(f) Single-headed inhibit arcs generated between the transitions and the places representing current and no current in circuit 1

(g) Double-headed arcs generated between the component state/mode place and the transition

(h) Double-headed arcs generated between the component state/mode place and the transition associated with the state

Figure 4.11: Procedure steps for the construction of a Petri net representing circuit 1
Figure 4.12: Petri net for current and no current in circuit 2

Figure 4.13: Petri net for the monitoring of the state of circuit 2 and the connection to the SPN
4.3. Petri Net Models

Construction Procedure

The procedure moves through the components in the order that they are given by the user. The procedure for the completion of the SPN is given as follows:

1. Check: Does component $A$ connect to another component/entity in the system structure?
   
   (a) Yes: continue to step 2.
   
   (b) No: This is going to the boundary out of the system structure. Create single headed arc from output place to a boundary line. Return to step 1 for next component.

2. Taking CPN of component $A$, identify output linked to an input of a component, $B$, using the topology information. Are the output/input types the same?
   
   (a) Yes: Continue to step 3.
   
   (b) No: This connection is labelled incorrectly or there is an error in the decision table for this component.

3. In the CPNs there will be an output place of component $A$ that can be linked to an input place of component $B$, e.g. \textit{"OUT = C"} and \textit{"IN = C"}. Merge the places in the Petri nets into a single place.

4. Are there any other outputs associated with component $A$?
   
   (a) Yes: Repeat steps 1-3 for new output connection.
   
   (b) No: Go back to step 1 for new component.

To demonstrate this procedure, the system in Figure 4.3 was used. Starting with the power supply, $PS_1$, as seen in Figure 4.14a, step one looks at the topology and identifies whether component $PS_1$ is connected to any other component in the system. $PS_1$ is identified to be connected to the component $FS$ (Figure 4.14b). Step two identifies which output place of the component $PS_1$ is connected to which component $FS$ input place. The values within the input and output places are checked to ensure that they have the same values, e.g. $C$ and $NC$. Step 3 merges the place output and input places as seen in Figure 4.14c. Step four looks to identify any other $PS_1$ outputs, as there are none the procedure moves on to the next component, $FS$. The procedure is repeated for each component in the system. These can be seen in Figures 4.14d-4.14f.
(a) Starting with the first component, the power supply $PS_1$

(b) Identifying the link between the power supply’s out and the next component in the system topology, the fuse $FS$

(c) Creating the arc between components $PS_1$ and $FS$

(d) Identifying the link between the fuse’s out and the next component in the system topology, the motor $M$

(e) The arc between components $FS$ and $M$ is created

(f) As the system topology identifies $PS_1$ as the component connected to $M$ and arc is created between $M$ component out and $PS_1$ component in

Figure 4.14: Procedure steps for the construction of the system Petri net
4.3.4 Phase Petri Nets

The main property of the phase Model is that this monitors the time of the simulation. Any components that are governed by a specified time are controlled by the Phase model. Although the phase transition table may not have all associated time frames where components change, the Phase model must still account for all possible combinations.

4.3.4.1 Phase Petri Net Construction

The PPN is constructed principally from the phase transition table. However, as the Phase Petri net controls components, when the component decision table contains a time heading, these need to be accounted for. Therefore, the second set of information comes from the component decision tables.

Construction Procedure

The procedure for generating the Petri nets to show the phases of the mission is detailed below:

1. Going through the phase transition table, identify the main phases using the time column. The times will exist in rows where a main phase is in the fromphase column. The times listed represent the phase length by taking the value in the time column minus any previous phase lengths identified. The symbol \( \delta \), which represents a small amount of time, is ignored in this step. For each of the phases identified and times listed in the table a place is created.

2. Between places representing the phase names/numbers and places representing a phase length, \( t \), create a timed transition with a delay of time \( t \).

3. Create a single headed arc between the place representing the phase name/number and the transition. Also create a single headed arc between the transition and the place representing the phase length, \( t \).

4. Between the places representing the phase length \( t \) and the next phase name/number in the list, create an immediate transition.

5. Create a single headed arc between the place representing the phase length, \( t \) and the immediate transition and create a single headed arc between the immediate transition and the place representing the next phase name/number.

6. Using the phase transition table, identify all types (phase names/numbers) in the From Phase and To Phase columns.
7. Apart from the phase names/numbers, found in step 1, create a place representing each phase name/number.

8. Moving through each row of the phase transition table:

(a) Create an immediate transition if:
   i. There is a time associated with the row i.e. not ‘-’ apart from δ. The transition is placed between the place representing the phase length (associated with the row) and the To Phase place.
   ii. Either the From Phase or To Phase is not one of the main phases identified in step 1. The immediate transition is placed between the FromPhase place and the ToPhase place.

(b) For any rows that contain δ a timed transition is created with a delay of δ and placed between the FromPhase place and the ToPhase place.

(c) Create a single-headed arc:
   i. Between the phase that either represents the phase length or the From Phase, depending on step 8(a).
   ii. Between the transition generated in step 8(a) and the ToPhase place.

(d) The condition column states which component and either the mode or output that can instigate the change in phase. Therefore a link is created between the CPN holding the place representing the component mode or output type and the transition representing the row. A double headed arc is used.

9. Finally, time columns in component decision tables are considered if there are any in the system.

(a) A list of the different times, or time frames, that exist in the component decision tables are generated.

(b) Create a place if the following are true:
   i. The time does not already exist as a place created in step 1.
   ii. If there are any component decision tables that include a time frame that the main phases effectively represent, e.g. 0< t< 1.5. If the main phase is length 1.5.

(c) When all places are created, an immediate transition is generated for each time frame in the list. An arc is created between the newly created transition and the place representing the time frame.

(d) Input places into the transition depends on the following:
4.3. Petri Net Models

i. If the time frame stated has a \( \leq \) sign associated with the first half of the time frame, then the input arc comes from the place that represents that instant in time.

ii. Else, the input place comes from the main phase that begins at the same time as the time frame.

(e) The arc from the place representing the time frame depends on the second half of the time frame:

i. If the second half of the time frame stated has a \( \leq \), then the arc connects to the same transition as the place that represents that specific moment in time, created in step 1.

ii. Else, the arc connects to the same transition as the one of the main phases that also has the same end of the time frame. This transition will be one that was created in step 3.

10. A connection is then made between the time frame places generated in step 9 and the component transition that has a time element associated with it. A double headed arc is used here.

To demonstrate the procedure the system in Figure 4.15 is used. This system has three phases; the first is a discrete phase which is the closing of the switch, the second phase the heater, \( H \), for a time, \( t_1 \) is on and the third phase the heater is off and the fan is turned on, for a length of \( t_2 \). The phase transition table for this system can be seen in Table 4.6. The nine phases listed in the table are as follows:

1. Start-up Phase
2. Heating Phase
3. Cooling Phase
4. System failed to start
5. System failed to heat for correct amount of time
6. System failed to cool for correct amount of time
7. System overheated due to heater on for too long
8. System overcooled due to fan on for too long
9. Mission Completed
All components are assumed to be in a working condition at the start of the mission and the heater, $H$, and the fan, $F$ are OFF. Timer relay, $TIM_1$, should last for a length of time, $t_1 + t_2$ and timer relays, $TIM_2$ and $TIM_3$, should last for a length of time, $t_1$.

The initial modes of the components are as follows:

- Toggle switch, $TS$, mode = OP
- Normally-open contact, $C_1$, mode = OP
- Normally-open contact, $C_2$, mode = OP
- Normally-closed contact, $C_3$, mode = CL
- Normally-open contact, $C_4$, mode = OP

The system is started by pressing the switch, $TS$, closed. This initiates the timer relay, $TIM_1$, which closes the contact $C_1$. By closing $C_1$ this completes the inner circuit, and starts timers $TIM_2$ and $TIM_3$. By starting timer relays $TIM_2$ and $TIM_3$ these close and open normally-open contact, $C_2$, and normally-closed contact, $C_3$, respectively. At this point the heater, $H$, is ON. Relay, $R$, is also energised and closes contact $C_4$. After a time $t_1$, the timer relay $TIM_2$ de-energises and opens normally-open contact $C_2$. This breaks the circuit to the heater, and turns the heater OFF. Also at time $t_1$, the timer relay $TIM_3$ de-energises and normally-closed contact, $C_3$, closes. This completes the circuit and the fan turns ON. After a time $t_1 + t_2$, the timer relay $TIM_1$ de-energises and opens the normally-open contact, $C_1$. As no current is flowing through the relay, $R$, this de-energises and opens contact $C_4$. Therefore there is no longer any current in the circuit with the fan, therefore turning the fan OFF. $S_1$ and $S_2$ are sensors for the components $H$ and $F$, respectively.

Table 4.6: Phase Transition Table for heater fan system

<table>
<thead>
<tr>
<th>$t$</th>
<th>From Phase</th>
<th>To Phase</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$\delta$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>$-$</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>$t_1$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>$t_1$</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>$-$</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>$t_1 + t_2$</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>$t_1 + t_2$</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>
Figure 4.15: System diagram and topology for a heater and fan system
Table 4.7: Decision Table for Timer Relay TIM1

<table>
<thead>
<tr>
<th>t</th>
<th>In State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 &lt; t &lt; t_1 + t_2</td>
<td>C W</td>
<td>EN</td>
</tr>
<tr>
<td>2</td>
<td>t_1 + t_2</td>
<td>C W</td>
<td>DE</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>NC</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.8: Decision Table for Timer Relays TIM2, TIM3

<table>
<thead>
<tr>
<th>t</th>
<th>In State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 &lt; t &lt; t_1</td>
<td>C W</td>
<td>EN</td>
</tr>
<tr>
<td>2</td>
<td>t ≥ t_1</td>
<td>C W</td>
<td>DE</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>NC</td>
<td>–</td>
</tr>
</tbody>
</table>

Following the steps listed the PPN for this system was generated using the phase transition table in Table 4.6 and the component decision tables. The component decision tables which incorporate time (the timer relays) are given in Tables 4.7 and 4.8. This procedure was split into three sections; identifying and creating the main phase places and connections, the phase transition table and the component decision tables. Figure 4.16 represents the main phases, steps 1-5. Figure 4.17 represents the phase transition table, steps 6-8. Figure 4.18 represents the component decision tables, steps 9-10.

Starting from the phase transition table, the main phases were identified as phase 1, 2 and 3, as these have a time associated with the row. The phase lengths were identified as t = 0, t = t_1 and t = t_2. This step can be seen in Figure 4.16a. Step two created a timed transition with a delay of the phase length for each of the phase lengths given. This can be seen in Figure 4.16b. Step three created single headed arcs between the places representing the phases and the transitions. A single-headed arc was also connected between the transition and the time place. This can be seen in Figure 4.16c. The next step created an immediate transition between the places representing the phase lengths and the next phase. This can be seen in Figure 4.16d. The final step in the first stage generated a single-headed arc between the places representing the phase lengths and the immediate transitions generated in the previous step. Finally, single-headed arcs were created between the immediate transitions and the next phase places.

The second stage used the phase transition table to first identify all phase names/numbers \{1, 2, 3, 4, 5, 6, 7, 8, 9\}. Once all were identified, as seen in Figure 4.17a, a place was created for each, except those that were already created in the first step, which can be seen in Figure 4.17b. Step eight moves through each row of the table, if a time exists in
Algorithm

4.4 Algorithm

The generation of the Petri nets discussed in Sections 4.3.1-4.3.4 are performed by the software. An overview of the design flow in the context of the program is presented below. The algorithm is presented in the flow chart in Figure 4.19.

1. The information required from the user is the following:

   (a) Information about the components in the system; how they work, fail and what type of connections each component has.

   (b) A system topology diagram of how the components link together.
Chapter 4. Modelling of Non-Repairable Systems

(a) Create a place for each phase and each phase length

(b) Create a timed transition with a delay of the phase length

(c) Create an arc between the phase places and the timed transition and an arc between the timed transition and the place representing the length of the phase

(d) Immediate transitions are created between the places representing the length of the phase and the next phase

(e) Create an arc between the places representing the length of the phase and the immediate transition and an arc between the transition and the next phase

Figure 4.16: Steps 1-5 of the construction procedure applied to the heater, fan system
4.4. Algorithm

<table>
<thead>
<tr>
<th>t</th>
<th>From Phase</th>
<th>To Phase</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>C1 MODE=CL</td>
</tr>
<tr>
<td>2</td>
<td>δ</td>
<td>4</td>
<td>C1 MODE=OP</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>5</td>
<td>S1 OUT=OFF</td>
</tr>
<tr>
<td>4</td>
<td>t1</td>
<td>2</td>
<td>S1 OUT=OFF</td>
</tr>
<tr>
<td>5</td>
<td>t1</td>
<td>2</td>
<td>S1 OUT=ON</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>5</td>
<td>S2 OUT=OFF</td>
</tr>
<tr>
<td>7</td>
<td>t1+t2</td>
<td>3</td>
<td>S2 OUT=ON</td>
</tr>
<tr>
<td>8</td>
<td>t1+t2</td>
<td>3</td>
<td>S2 OUT=OFF</td>
</tr>
</tbody>
</table>

(a) Identifying all phase names/numbers in the phase transition table

(b) Generating a place for each of the phase names/numbers identified in step 6

(c) Creating the transitions, arcs using the rows of the phase transition table

Figure 4.17: Steps 6-8 of the construction procedure applied to the heater, fan system
(a) Time frames identified in the component decision tables have a place generated

(b) Links to the component Petri nets is established

Figure 4.18: Steps 9-10 of the construction procedure applied to the heater, fan system
(c) A phase transition table of the mission. The user is required to know how many phases the system can reside in and how the system can change from one phase to another.

(d) Component failure rates, or failure distributions.

(e) Initial component modes for components that have multiple modes of operation.

(f) Initiating component information.

2. This information is presented in five types of files:

(a) Text-based decision tables for components that don’t already exist in the component library.

(b) Text-based operational mode tables for components that don’t already exist in the component library.

(c) A text-based description of the system topology, listing instances of components and their connections.

(d) A text file containing the phase transition table for the system.

(e) A text file containing the simulation information including the failure data, initial operating modes and the initiating component.

3. The component decision tables and operational mode tables are added to the component library in the system. This library is persistent and allows re-use of components in the future.

4. The circuit lists are automatically generated by the software using the information in the component tables within the library and the system topology.

5. The following Petri nets are now automatically generated by the software:

(a) **CPNs**: Generated for every type of component in the system found in the component library. These are stored in a CPN library.

(b) **CiPNs**: Generated for every list found by the software.

(c) **SPN**: For every instance of a component listed in the system topology file, there is a copy of the CPN made from the CPN library. With the information in the system topology file, the CPNs are connected together. The CiPNs and the CPNs make up the SPN. As the CPN library holds the generic CPNs, the component failure data is only added within the SPN. The component failure data is given by the user in the form of a text file.

(d) **PPN**: This is generated from the phase transition table and the component decision tables found in the component library.
6. The reliability model is the SPN and the PPN connected together. The last piece of information required from the user is the initial conditions which contains the following:

(a) Initial modes of components that have operational mode tables.
(b) The initiating component with the output/input and the value.

7. Once the initiating condition is set then the simulation can begin.

8. After the software has run information about the reliability/unreliability of the system undertaking the mission specified will be given to the user.

4.5 Summary

The procedures demonstrated here form the basis of the algorithms for the software created. These procedures could also be used by individuals to generate the reliability models by hand as they are simple and efficient. They show a reliable method of taking the information held within the text-based files and generating the reliability models from them.
Figure 4.19: Flow chart of the algorithm
Chapter 5
Application of the Procedure to Pressure Tank System

Contents

5.1 Introduction .............................................. 123
  5.1.1 The Pressure Tank System ......................... 124

5.2 System and Mission Description ....................... 125
  5.2.1 Components ........................................ 125
  5.2.2 System Structure .................................... 130
  5.2.3 Circuits ............................................ 130
  5.2.4 Mission Profile .................................... 131

5.3 Pressure Tank System Model Construction .......... 131
  5.3.1 Component and System Petri Nets .................. 132
  5.3.2 Circuit Petri Nets .................................. 132
  5.3.3 Phase Petri Net .................................... 134
  5.3.4 The Completed Model ................................ 142

5.4 Summary .................................................. 142

5.1 Introduction

The procedures discussed in Chapter 4 are brought forward to be demonstrated using a pressure tank system. This pressure tank system is a simple system with few phases, making it an ideal system to demonstrate the procedure. This chapter discusses the pressure tank system in terms of the process the system undertakes, for the purposes of understanding how the system can move from one phase to another. There are two distinct sections of this chapter. The first section describes each element of the system: the components, the circuits and the structure. This section also describes each of the phases the system can reside in through the mission. The second section takes these descriptions and, using the procedure outlined in Chapter 4, shows the final Petri nets:
5.1.1 The Pressure Tank System

The pressure tank system as a schematic can be seen in Figure 5.1. This shows all the individual components in the system and how they connect to each other.

5.1.1.1 System Process

The system is initially started by depressing switch $S_1$, momentarily applying power to the timer relay, $TIM$, whose contacts close and start the timer. Note that switch $S_2$ is closed at system startup. Switch $S_1$’s contacts open. Power is applied to relay $R$ whose contacts close and start the pump motor. The tank starts to fill. After a time $t_1$ the timer relay contacts open, relay $R$ de-energises and its contacts open, thus removing power from the pump motor. When $TIM$ is de-energised, the timer clock resets. The operator will notice the tank pressure by the pressure gauge and will open the valve to empty the tank. After a time $t_2$, the tank will have emptied sufficiently for filling to start again by the operator closing the valve and depressing switch $S_1$. Switch $S_2$ is a safety mechanism built into the system so that, in the event that a failure occurs and the tank overfills, the operator, who will be alerted by the pressure gauge, can stop the pump by opening that switch, cutting the component Petri nets, the circuit Petri nets, the system Petri net and the Phase Petri net.
power to $R$.

### 5.1.1.2 Initial Conditions of the System

All components are assumed to be in a working condition and that the following components exist in these initial modes:

- Switch $S_1$ is open.
- Switch $S_2$ is closed.
- Tank is empty.
- Relay contact, $RC$ is open.
- Timer relay contact, $TC$ is open.
- Valve $V$ is closed.

### 5.2 System and Mission Description

This section discusses the components within the pressure tank system and how each of these components behaves through the decision tables and, when applicable, operational mode tables. Electrical circuits present within the system are listed. The system itself is described in terms of its structure and the mission is described through a phase transition table representing the process described in Section 5.1.1.1.

#### 5.2.1 Components

The components of a non-repairable system are described by the following information:

- Decision Tables
- Operational Mode Tables
- Failure Data

Each component in the pressure tank system is discussed in detail below.

#### 5.2.1.1 Decision and Operating Mode Tables

The component tables describing the pressure tank system can be seen in Tables 5.1-5.15. Tables 4.1-4.3, which showed the component tables for a power supply and toggle switch,
are re-used here to represent $PS_1$, $PS_2$ (power supplies) and $S_2$ (toggle switch). As push-switch $S_1$ behaves differently to $S_2$, this component requires a different set of component tables.

Table 5.1: Operational mode table for push switch $S_1$

<table>
<thead>
<tr>
<th>Mode 1</th>
<th>Command (In1)</th>
<th>State</th>
<th>Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Closed</td>
<td>FCL</td>
<td>Closed</td>
</tr>
<tr>
<td>2</td>
<td>Closed</td>
<td>CL</td>
<td>Closed</td>
</tr>
<tr>
<td>3</td>
<td>Closed</td>
<td>NA</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>Open</td>
<td>–</td>
<td>FOP</td>
</tr>
<tr>
<td>5</td>
<td>Open</td>
<td>CL</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>NA</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 5.2: Decision table for switches $S_1$ and $S_2$, and contacts $TC$ and $RC$

<table>
<thead>
<tr>
<th>In 2</th>
<th>Mode</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>NC</td>
</tr>
</tbody>
</table>

Table 5.3: Operational mode table for toggle switch $S_2$ and Valve $V$

<table>
<thead>
<tr>
<th>Mode 1</th>
<th>Command (In1)</th>
<th>State</th>
<th>Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Closed</td>
<td>FCL</td>
<td>Closed</td>
</tr>
<tr>
<td>2</td>
<td>Closed</td>
<td>CL</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>Closed</td>
<td>OP</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>Closed</td>
<td>NA</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>Open</td>
<td>–</td>
<td>FOP</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>OP</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>Open</td>
<td>CL</td>
<td>W</td>
</tr>
<tr>
<td>8</td>
<td>Open</td>
<td>NA</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 5.4: Decision table for power supplies $PS_1$ and $PS_2$, and fuse $FS$

<table>
<thead>
<tr>
<th>In</th>
<th>State</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 5.5: Decision table for relay $R$

<table>
<thead>
<tr>
<th>In</th>
<th>State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>W EN</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>F DE</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>-</td>
<td>DE NC</td>
</tr>
</tbody>
</table>

Table 5.6: Decision table for timer relay $TIM$

<table>
<thead>
<tr>
<th>$t$</th>
<th>In</th>
<th>State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t &lt; t_1$</td>
<td>C</td>
<td>W EN</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>$t \geq t_1$</td>
<td>C</td>
<td>W DE</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>F</td>
<td>DE</td>
<td>NC</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>NC</td>
<td>-</td>
<td>DE NC</td>
</tr>
</tbody>
</table>

Table 5.7: Operational mode table for timer relay contact $TC$ and relay contact $RC$

<table>
<thead>
<tr>
<th>Mode 1</th>
<th>Command (In1)</th>
<th>State</th>
<th>Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Closed</td>
<td>–</td>
<td>FCL</td>
</tr>
<tr>
<td>2</td>
<td>Closed</td>
<td>EN</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>Closed</td>
<td>DE</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>Open</td>
<td>–</td>
<td>FOP</td>
</tr>
<tr>
<td>5</td>
<td>Open</td>
<td>DE</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>EN</td>
<td>W</td>
</tr>
</tbody>
</table>

Table 5.8: Decision table for junctions $J_1$ and $J_3$

<table>
<thead>
<tr>
<th>In 1</th>
<th>In 2</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>NC</td>
</tr>
</tbody>
</table>

Table 5.9: Decision table for junctions $J_2$ and $J_4$

<table>
<thead>
<tr>
<th>In 1</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>NC</td>
<td>NC</td>
</tr>
</tbody>
</table>
Table 5.10: Decision table for motor $M$

<table>
<thead>
<tr>
<th>In</th>
<th>State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>W</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>F</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>–</td>
<td>NC</td>
</tr>
</tbody>
</table>

Table 5.11: Decision table for pump $P$

<table>
<thead>
<tr>
<th>In</th>
<th>State</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ON</td>
<td>W</td>
<td>FL</td>
</tr>
<tr>
<td>2 –</td>
<td>F</td>
<td>NFL</td>
</tr>
<tr>
<td>3 OFF</td>
<td>–</td>
<td>NFL</td>
</tr>
</tbody>
</table>

Table 5.12: Decision table for tank $T$

<table>
<thead>
<tr>
<th>t</th>
<th>In 1</th>
<th>In 2</th>
<th>State</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>FL</td>
<td>Open</td>
<td>W</td>
<td>CONST</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>FL</td>
<td>Closed</td>
<td>W</td>
<td>INC</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>NFL</td>
<td>Closed</td>
<td>W</td>
<td>CONST</td>
</tr>
<tr>
<td>4</td>
<td>$t \leq t_1$</td>
<td>NFL</td>
<td>Open</td>
<td>W</td>
<td>CONST</td>
</tr>
<tr>
<td>5</td>
<td>$t_1 &lt; t \leq t_1 + t_2$</td>
<td>NFL</td>
<td>Open</td>
<td>W</td>
<td>DEC</td>
</tr>
<tr>
<td>6</td>
<td>$t \leq t_1$</td>
<td>–</td>
<td>–</td>
<td>F</td>
<td>CONST</td>
</tr>
<tr>
<td>7</td>
<td>$t_1 &lt; t \leq t_1 + t_2$</td>
<td>–</td>
<td>–</td>
<td>F</td>
<td>DEC</td>
</tr>
</tbody>
</table>

Table 5.13: Decision table for pressure gauge $PG$

<table>
<thead>
<tr>
<th>t</th>
<th>In</th>
<th>State</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t &lt; t_1$</td>
<td>CONST</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>$t &lt; t_1$</td>
<td>INC</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>$t_1$</td>
<td>CONST</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>$t_1$</td>
<td>INC</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>DEC</td>
<td>W</td>
</tr>
<tr>
<td>6</td>
<td>$t_1 &lt; t \leq t_1 + t_2$</td>
<td>CONST</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>$t_1 &lt; t \leq t_1 + t_2$</td>
<td>INC</td>
<td>W</td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>–</td>
<td>F_LOW</td>
</tr>
<tr>
<td>9</td>
<td>–</td>
<td>–</td>
<td>F_HIGH</td>
</tr>
<tr>
<td>10</td>
<td>–</td>
<td>–</td>
<td>F_VHIGH</td>
</tr>
</tbody>
</table>
Table 5.14: Decision table for operator $OP$

<table>
<thead>
<tr>
<th>t</th>
<th>In 1</th>
<th>State</th>
<th>Out 1</th>
<th>Out 2</th>
<th>Out 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>LPR</td>
<td>W</td>
<td>CL</td>
<td>CL</td>
</tr>
<tr>
<td>2</td>
<td>$0 &lt; t &lt; t_1 + t_2$</td>
<td>LPR</td>
<td>W</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>HPR</td>
<td>W</td>
<td>OP</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>VHPR</td>
<td>W</td>
<td>NA</td>
<td>OP</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>F</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 5.15: Decision table for valve $V$

<table>
<thead>
<tr>
<th>In 2</th>
<th>Mode</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>Closed</td>
</tr>
<tr>
<td>2</td>
<td>NFL</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>FL</td>
<td>Open</td>
</tr>
</tbody>
</table>

5.2.1.2 Component Failure Data

The failure data for each component in the pressure tank system is listed within Table 5.16.

Table 5.16: Pressure tank system component failure data

<table>
<thead>
<tr>
<th>Component identifier</th>
<th>Failure Mode</th>
<th>Failure Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>F_closed</td>
<td>0.1</td>
</tr>
<tr>
<td>S1</td>
<td>F_open</td>
<td>0.1</td>
</tr>
<tr>
<td>S2</td>
<td>F_closed</td>
<td>0.8698</td>
</tr>
<tr>
<td>S2</td>
<td>F_open</td>
<td>0.0001</td>
</tr>
<tr>
<td>PS1</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>PS2</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>CT</td>
<td>F_closed</td>
<td>0.1</td>
</tr>
<tr>
<td>CT</td>
<td>F_open</td>
<td>0.1</td>
</tr>
<tr>
<td>CR</td>
<td>F_closed</td>
<td>0.00023</td>
</tr>
<tr>
<td>CR</td>
<td>F_open</td>
<td>0.00023</td>
</tr>
<tr>
<td>TIM</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>R</td>
<td>F</td>
<td>0.1</td>
</tr>
<tr>
<td>M</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>FS</td>
<td>F</td>
<td>0.01</td>
</tr>
<tr>
<td>P</td>
<td>F</td>
<td>0.1</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>0.0001</td>
</tr>
<tr>
<td>V</td>
<td>F_closed</td>
<td>0.03</td>
</tr>
<tr>
<td>V</td>
<td>F_open</td>
<td>0.03</td>
</tr>
<tr>
<td>PG</td>
<td>F_LOW</td>
<td>0.01</td>
</tr>
<tr>
<td>PG</td>
<td>F_HIGH</td>
<td>0.01</td>
</tr>
<tr>
<td>PG</td>
<td>F_VHIGH</td>
<td>0.01</td>
</tr>
<tr>
<td>OP</td>
<td>F</td>
<td>0.1</td>
</tr>
</tbody>
</table>
5.2.2 System Structure

The system description requires the following elements:

- **Component Descriptions:** Using the component decision tables to identify the number of input and outputs.

- **Schematic of the system:** How the components link together showing the relationship between one component’s output and another component’s input.

The system topology diagram (or schematic) of the pressure tank system can be seen in Figure 5.2.

5.2.3 Circuits

If a system contains electrical components then there will be circuits within that system. These must be identified as current flowing in a circuit depends on all its components simultaneously; identifying circuits simplifies the analysis of these components. Within the pressure tank system five circuits were identified as seen below:

1. \{PS1, S2, J2, TIM, J3, J4, S1, J1, PS1\}
2. \{PS1, S2, J2, TIM, J3, J4, CT, J1, PS1\}
3. \{PS1, S2, J2, R, J3, J4, CT, J1, PS1\}
4. \{PS1, S2, J2, R, J3, J4, S1, J1, PS1\}
5.3. Pressure Tank System Model Construction

5. \{PS2, FS, CR, M, PS2\}

The above list of circuits begin and end with the same component, in this case the power supply, to show that a continuous loop must be present for it to be a circuit. This list of circuits would be used to generate the CiPNs as discussed in 4.3.2.

5.2.4 Mission Profile

The mission profile is represented through a phase transition table, but before it can be constructed all phases that a system can enter must be identified. The pressure tank system has four phases and are listed as follows:

- Phase 1: Start up, a discrete phase, only occurring momentarily when switch S1 is pressed at t=0
- Phase 2: Filling of the tank, T, with duration \(t_1\)
- Phase 3: Opening the valve, V, a discrete phase at \(t=t_1\).
- Phase 4: Emptying the tank, T, with duration \(t_2\)

These four phases would be the normal mission for the pressure tank system as long as no component or combination of components cause the system to fail. Additional phases, representing different failures, must also be identified. The phases representing a failure of the pressure tank system are shown below:

- Phase 5: System failure due to overfill
- Phase 6: System overfill with system shutdown (aborted by operator using switch S2)
- Phase 7: System overfill with failure to shutdown system (abort failed)
- Phase 8: System failure not from overfill (unrelated component failure)

Finally, a phase is designated as mission success and in this case this was assigned to Phase 9.

5.3 Pressure Tank System Model Construction

This section shows how the information captured in the previous section is represented as the Petri Nets described in Chapter 4: CPNs, CiPNs, SPN and PPN.
5.3.1 Component and System Petri Nets

Using the procedure in Section 4.3.1.1 and the component tables listed in Section 5.2.1.1, the CPNs were constructed for each component type. Figure 5.3b and Figures 5.6-5.8 show the constructed CPNs for the pressure tank system. The dashed lines connecting to the transitions of the Petri nets seen in Figures 5.5, 5.7a, 5.8b, 5.8c and 5.8d are the connections that link to the PPN.

Figure 5.4 shows an example of a Petri net for a component with multiple modes of failure, but only a single failure rate which is common to all failure modes. Examples of such components are S1, S2, TC, RC and V.

The pressure gauge, PG, has multiple failure states and only one operational mode. This component requires a different set of working-to-failed transitions. This is shown in Figure 5.5.

The operational mode table and decision table equivalents of each of the components in the pressure tank system can be seen in Figures 5.6 - 5.8.

The system Petri net is created by using the system schematic/topology diagram. For every instance of a component type found within the diagram, an instance of the equivalent Component Petri net is used. Identifying the relationships between one component’s output and another component’s input connects the component instance places together. The system Petri net for the pressure tank system can be seen in Figures 5.9-5.12.

5.3.2 Circuit Petri Nets

The circuit Petri nets are created directly from the circuit lists and within the pressure tank system five circuits were identified as listed in Section 5.2.3. Each of the lists is used to create an individual circuit Petri net. The equivalent circuit Petri nets for the listed circuits can be seen in Figures 5.13 and 5.14. The place that represents each of the components in the list is dependent on whether the component has a single or multiple mode(s) of operation. If the component has a single mode of operation, then the component’s state,
5.3. Pressure Tank System Model Construction

Figure 5.3: CPN construction and integration of the component tables for the component push-switch (S1)

i.e. working or failed, is used to represent the component as this dictates if the component can allow current or no current to pass through it. If the component has multiple modes of operation then the component’s mode is used to represent the component as this will dictate the potential current condition or no current condition.

The circuit Petri nets link, via a specific component in each circuit, to the system Petri net. The component in each circuit that makes this link is identified by an algorithm that follows component inputs and outputs of type (C, NC) from the initiating component. The first component encountered that has more than one operational mode is selected. If no such components are found then the first circuit component encountered by the algorithm is selected instead. The preference for components with multiple modes of operation reduces the propagation time of changes between modes in circuit components. The circuits seen in Figure 5.13a and Figure 5.13d are connected through component S1, circuits in Figure
Chapter 5. Application of the Procedure to Pressure Tank System

Figure 5.4: Component Petri net of the failure rate for components $S_1$, $C_R$, $C_T$ and $V$

Figure 5.5: Pressure Gauge, $PG$, transitions between the working state and the different failure states

5.13b and Figure 5.13c are connected through TC, and circuit in Figure 5.14 is connected through component $RC$. The method of connection can be seen in Figure 5.15.

5.3.3 Phase Petri Net

The PPN is created in a similar fashion to the decision tables and operational mode tables in that each row of the table is represented by a unique transition within the PPN. However, due to the time element of the PPN, the overall construction process is a little more complex. As the PPN effectively manages the time element of the mission, there are a number of connections to components within the pressure tank system from the PPN. These connections enable phase-based transitions within components to fire.

The phase transition table that is detailed in Table 5.17 can be mapped to the the PPN in Figure 5.16.
5.3. Pressure Tank System Model Construction

(a) Switch S2 Petri net

(b) Contacts $C_R$ and $C_T$ Petri net

(c) Valve $V$ Petri net

Figure 5.6: Petri nets for components with multiple modes of operation
Figure 5.7: Petri nets for components within circuits
5.3. Pressure Tank System Model Construction

Figure 5.8: Component Petri nets for non-circuit components
Chapter 5. Application of the Procedure to Pressure Tank System

Figure 5.9: Part one of the system PN
Figure 5.11: Part three of the system PN
5.3. Pressure Tank System Model Construction

Figure 5.12: Part four of the system PN.
5.3.4 The Completed Model

Petri nets have now been generated for each component and assembled into the system Petri net, following the system structure. Circuit Petri nets, representing electrical circuits in the system, have been generated and connect to the system Petri net. Finally, a phase Petri net, dealing with the discrete stages of the mission, has been generated and connected. This forms a single large Petri net to model the overall phased mission system.

5.4 Summary

This chapter demonstrates how each of the construction procedures described in Chapter 4 were applied to a defined system and mission. Each of the different Petri nets shown here was generated using these procedures. Although these were generated manually, it will be shown in the following chapter how this process can be automated using software that reads text-based input files. In the next chapter this system and the Petri nets shown here are used to validate the software’s capabilities.
Figure 5.13: Circuit Petri Nets for Circuits 1 to 4 of the Pressure Tank System
Figure 5.14: Circuit Petri Net for Circuit 5 of the Pressure Tank System
Figure 5.15: Circuit Petri net linkage Petri nets
Chapter 5. Application of the Procedure to Pressure Tank System

Figure 5.16: Phase Petri net for the Pressure Tank System
Chapter 6

Automated Reliability Modelling

Contents

6.1 Introduction .................................................. 147
   6.1.1 Object-Oriented Programming in C++ ...................... 148
   6.1.2 Key Definitions ............................................. 148
6.2 Software Files ............................................... 149
   6.2.1 Component Description Files .............................. 150
   6.2.2 System Topology Description .............................. 152
   6.2.3 Mission Description ......................................... 153
   6.2.4 Simulation File ............................................. 155
   6.2.5 Setup File .................................................. 157
6.3 Software Structure .......................................... 158
   6.3.1 Storage of System and Mission Description ............... 158
   6.3.2 Building the Petri Net Model ............................. 171
   6.3.3 Simulating the Petri Net Model ........................... 178
6.4 Testing and Validation ...................................... 187
   6.4.1 Validation using Phase Fault Trees ....................... 188
6.5 Summary ...................................................... 196

6.1 Introduction

This chapter concerns the automation of the process discussed in chapters 4 and 5. The chapter begins with the files used to describe the system and the mission to be undertaken. Discussion is made relating to how the software extracts the different pieces of information from each of the file types entered. The chapter then moves on to the construction of the Petri net: how the information given in the input files is translated into the model used to simulate the system and mission. The next section gives details of the simulation process, from the initial conditions to any changes made to the model through the simulation process and then the results obtained at the end. The way in which the results are represented is also discussed at the end for completeness. The final section of the chapter details the
testing of the software and provides validation of the model through the testing of the pressure tank system described in Chapter 5.

6.1.1 Object-Oriented Programming in C++

The programming language chosen, C++, is an object-oriented programming language. Object-oriented programming uses the concept of a class to describe an object. A class is a module of code that contains functions and data in the form of variables. Other parts of the code interact with the class to access its data and perform its functions. This abstracts the complexity of the functions and data within, a concept known as encapsulation. For instance if a library were modelled as a class, some of the variables could be the name of the library, a list of books held by the library and the membership details of those that use the library. A function of the library could be to lend a book, or to enrol a new member. Books and members contained within the library could also be classes. Using classes as part of an object-oriented programming language aids in structuring large software projects such as the one discussed here and references will be made to this concept throughout the chapter.

6.1.2 Key Definitions

There are a number of phrases that will be used continually throughout this chapter and have been listed here for convenience.

- **port** - A component port is an input or output of a component.
- **link type** - This refers to the type of connection between components, e.g. circuit connection $C, NC$.
- **link name** - This is the name given to refer to a link between two connected components in the system topology.
- **class** - A module of code that encapsulates a set of related functions and data.
- **function** - Performs a defined process which operates on input data to provide output data.
- **variable** - Stores data of a certain type.
- **object** - An instance of a class.
- **string** - A collection of characters (letters, numbers and symbols).
- **vector** - A container for a series of variables of the same type. Vectors can be nested within one another to form multi-dimensional series of variables.
6.2 Software Files

The description of the system and mission are written in text files which are processed by the software. A single file is provided to the software, the *project file*, which contains a list of all files required to describe the system and the mission to be undertaken. This project file has a file extension of *.prj* to identify it as such. Within the project file the other files must be listed in a given order, as seen below:

1. Component Decision and Operational Mode Tables (file extensions *.dt* and *.omt*, respectively)
2. System Topology (file extension *.ss*)
3. Phase Transition Table (file extension *.ptt*)
4. Simulation Information (file extension *.sim*)

Each file type has a unique file extension in order to differentiate between them.

The project file is designed to contain only one system and one mission, and therefore should never contain another project file. The files contained within the project file are given in the order above due to numerous checks carried out by the software. The component files are listed first to identify the components to be used in the current system. The system topology file includes the list of connections between instances of the components, therefore if a component is listed in the system topology file that was not included in the component files an error would be issued to the user. The same would occur if a component was listed in the phase transition table file and was not included in the list of component files. The software cannot continue until all necessary files are added. Each file contains a single entity (i.e. one component) to aid clarity.

The fourth file type, *simulation information*, includes the following:

1. Initial component modes
2. Failure data
3. Initiating component

The initial component modes are listed in terms of the modes listed within the table files (DT and OMT). The name for the initial mode must match exactly with a mode contained in the DT or OMT. The failure information is given by using one of the following:

1. Exponential distribution (parameter: mean time to failure)
2. Weibull distribution (parameters: characteristic life and shape parameter)
3. Normal distribution (parameters: standard deviation and mean time to failure)

4. Time to failure

The first three points are the distributions that can be used to describe how a component fails and the fourth gives the user the ability to let a component fail at a specific time.

The last item included within the simulation file is the initiating component which, as it states, initiates the simulation process.

The following sections look at each of the file types: component tables, system topology, phase transition table and simulation information, discussing in detail the contents of each file.

6.2.1 Component Description Files

The component description files include the decision and operational mode tables discussed in Section 4.2.1.

6.2.1.1 Decision Table Files

For each type of component that exists within the system structure, a DT is required. A table is not required for each instance of the component as each instance of a component identified within the system topology file uses the same DT.

File Layout

The DT file includes the file identifier, component type and the associated DT. An example of a single mode component, a power supply, can be seen in Figure 6.1. A general line-by-line breakdown of the decision table file is given below:

- Line 1: File identifier, ‘DT’, followed by the component type, which must be unique
- Line 2: Open curly brace, ‘{’, to signify the beginning of the decision table.
- Line 3: The headings of the decision table:
  - Input headings: \textit{in : in1, in : in2, ... , in : inM}, where \(M\) is the total number of inputs into the component.
  - State heading or mode heading: \textit{state : state1} or \textit{mode : mode3}
  - Output headings: \textit{out : out1, out : out2, ... , out : outN}, where \(N\) is the total number of outputs from the component. Once all outputs have been identified, a semi-colon is used to signify the end of the headings.
6.2. Software Files

```
DT power_supply
{
in:in1, state:state1, out:out1;
C, W, C;
-, F, NC;
NC, -, NC;
}
```

Figure 6.1: Decision table file input format for single mode components

- Lines 4-P: \((P\) is dependent on the number of rows in the decision table.) This contains the decision table, using commas to signify the next column and a semi-colon to signify the end of the row.

- Line P+1: Closing curly brace, ‘{’ , to signify the end of the decision table.

The format described above must be adhered to otherwise the software will not allow the entry of the file and halt any further processing of the files.

6.2.1.2 Operational Mode Table Files

When a component has multiple modes of operation, an OMT file must be created in addition to the DT file. Similar to the decision table file, an identifier is used to show what type the file is, followed by the component type and the OMT.

File Layout

The OMTs are presented in the format seen in Figure 6.2, and described below:

- Line 1: File identifier, ‘OMT’, followed by the component type. The component type field acts as a unique identifier to relate an OMT to its corresponding DT.

- Line 2: Open curly brace, ‘{’ , to signify the beginning of the component information.

- Line 3: the headings of the operational mode table:
  - Starting mode: mode : mode1
  - Command: This will be the input from an outside influence, such as an operator. Instead of the word command it is written as in : inS, where \( S \) is the input connection number.
  - State heading: state : state1
  - Final mode: mode : mode2. After this mode a semi-colon is used to signify the end of the headings.
6.2.2 System Topology Description

The system topology file lists the connections between each component in the system. Component instances are listed, with each instance including a list of that component’s ports. Each port is assigned a link name; these names are arbitrary and are used to connect the ports of different components together. There should only be one file for the system topology and this should include every component within the system.

6.2.2.1 File Layout

The system topology file follows the following format:

- Line 1: File identifier, SS, followed by the name of the topology/system.
- Line 2: The word ‘begin’ should appear on the next line to signify the beginning of the topology.
- Line 3: Following the word begin is the first of the components in the topology. Each of the components is structured in the following way:

  - The first line of the new component takes the following format: ‘component_identifier : component_type’, for example PS1 : power_supply. The component identifier is arbitrary and unique to that component.
6.2. Software Files

- The next line denotes the beginning of the port list, in which ports are mapped to links: port map(

- The next lines show the connections to each of the components ports (in, out). For each port the following syntax is used: in1 => link_one; and out1 => link_two; This is completed for each of the component’s ports. link_one and link_two are examples of arbitrary link names. These same names are then used to connect these ports to the ports of another component.

- When all the ports have been considered, the next line ends with ‘)’ to signify the end of the component’s ports.

- Last line: Once all the components have been considered the file ends with the word ‘end’ to signify the end of the topology for the system.

Each link between component ports requires a unique name. In the example given in Figure 6.3, the link names use a naming convention related to the component instance identifiers. The identifiers can be whatever the user chooses, as long as the link name is repeated only once with the connected component. For example in Figure 6.3 component S1, port Out1 shares the same link name (S1_J1) with the component J1, port In2 as these components are connected through these ports. Note that to connect one output to multiple component inputs, a junction component is required.

6.2.3 Mission Description

File Format

The description of the mission, the phase transition table as discussed in Section 4.2.4, is described in the phase transition table file (.ptt). An example of such a file can be seen in Figure 6.4. A line-by-line breakdown of the general format of this file type can be seen below:

- Line 1: File identifier, ‘PTT’, followed by the name of the mission.
- Line 2: Open curly braces ‘{’ to signify the beginning of the phase transition table.
- Line 3: Table headings time, from_phase, to_phase, condition;
- Lines 4-P: (P is dependent on the number of rows in the phase transition table). These lines contain the phase transition table itself. Each row of the transition table is formatted as follows:

  - The ‘time’ at which the transition can occur, which can be presented in three forms:
Figure 6.3: Example topology file format
Figure 6.4: Example mission file format

* A literal time, such as 0.
* An inequality in terms of t, specifying a range of times. This must be written in brackets; for example \((3 \leq t < 5)\)
* No time value, signified by ‘–’. This means the phase transition is not time-dependent.

- The ‘from_phase’ is a number or word that signifies which phase this transition is moving from.
- The ‘to_phase’ is a number or word that signifies which phase this transition is moving to.
- The ‘condition’ information is comprised of three elements: the component identifier, the component port and the value in the port. This is represented by the following format: 
  \[
  \text{component\_identifier.port\_name = port\_value;},
  \]
  for example 
  \[
  PS1.out1 = C;
  \]
  The component identifier states which instance of the component can cause the transition, followed by a full stop. This full stop signifies the port of this component will follow. The equals sign shows which value within that port will cause the transition. The semi colon signifies the end of the row.

- Line P+1: When the table is complete a closing curly brace \(\{\}\) is used.

6.2.4 Simulation File

The simulation file holds information related to each of the components in the system. This information includes the following information:

1. Initial operational modes for multiple operational mode components
2. Failure distribution data for each component

3. Initiating component

In order to simulate the system, initial operational modes must be stated for any components with multiple modes of operation. There should be failure data for each component, but if it is desired to test one part of the system assuming one or more components never fail, then this is catered for. When detailing the failure data of each component, if a component has multiple failure modes then each failure mode must be considered and listed. If the failure of a component is not affected by the mode that the component is in, then if each failure mode listing is the same the software will handle this. The initiating component is the component, port and port value that can start the simulation. An example of this file can be seen in Figure 6.5.

**File Format**

The simulation file has the following structure:

- The file identifier of ‘SIM’ is found at the top of the file.
- The initial component modes are listed as follows:

  1. Starting with the opening line `COMPONENT MODES {`
  2. List each component with multiple modes in the following format: ‘component_identifier mode = mode_value;’. 
3. This list is completed by including ‘\}' to signify that all component modes have been identified.

- The failure distributions for each component (or each component mode) are then listed in the following format:

  1. Starting with the opening line *FAILURE* \{
  2. For each component listed, the following format is used: ‘component_identifier failure_value(or mode) failure_distribution(parameter(s));’.
  3. This section finishes with ‘\}'.

- The last information held within the file is the initiating component. The following format is used:

  1. This starts with the opening line *INITIAL* \{
  2. There is only one component listed as the initiating component as follows: ‘component_identifier port_name = port_value;’.
  3. This section is completed with ‘\}'.

### 6.2.5 Setup File

The setup file includes configuration variables for the software. It is always included in projects and does not need to be listed in the `.prj` file. The setup file specifies the values used in electrical wires for current/no current and the value used to denote the working case within all decision and operational mode tables. This information defaults to the following:

- Current: *C*
- No Current: *NC*
- Working State: *W*

This setup file can be altered by the user if variations are required, but this is not recommended.

**File Format**

The file takes the following structure:

- The file identifier *INI* at the top of the file

- The working state value used in the tables is given next by the line ‘working state :: W; ’
• The current, no current values used in the tables are given in the line ‘wire type :: C, NC;’. The values are listed specifically in the order of current and then no current.

When all files are created and the name of each file is included in the project file, the user is ready to add their system to the software. Section 6.3 introduces the structure and how the information from all file types described above are dealt with and how this information is turned into a working model for the purpose of simulation.

6.3 Software Structure

The structure of the software can be split into three sections:

1. File parsing
2. Model creation
3. Simulation

The first step interprets the files discussed in the previous section and stores their contents in memory. The next section discusses this interpretation process. The second step takes the information from the files and generates a working model. The process of model construction concerns the following Petri nets, as discussed in chapter 4: component Petri nets (Section 4.3.1), circuit Petri nets (Section 4.3.2), system Petri net (Section 4.3.3), and phase Petri net (Section 4.3.4). The third step uses the model to simulate the system over a specified time for a given mission.

6.3.1 Storage of System and Mission Description

This section describes how the files created in the previous section are interpreted by the software and then stored to be used to build the model in the software.

Within the software there are a number of classes used to store the information held within the different files discussed above. For parsing the files and storing the information the primary classes used are fileHandler, topSystem and compLib. Each has a specific role within the software. The first, fileHandler, manages access to files and performs file parsing operations. Functions within this class are used to collect the information from the files listed within the project file. Depending on each file type (extension), the class executes a different set of functions. The topSystem class holds all the information that the fileHandler class extracts from the files. The fileHandler class dictates where in the topSystem class the information is held. The compLib class is the component library which holds all the individual component types found in the component decision and operational
6.3. Software Structure

Figure 6.6: Class view of topSystem, fileHandler and compLib

mode tables. The information held within the component library is later used to generate instances of the components.

A class representation of each of the above can be seen in Figure 6.6.

6.3.1.1 Storage of Component Descriptions

This section describes how the information in the component files is extracted by the software and stored as useful information in preparation for the model generation. Both the decision table and operational mode table files are considered here.

Figure 6.7 shows each class used when a component object is created. The figure shows how the other classes relate to the component class. At this point in the software process, the only class that would not be directly called on is the failRepair class. This is dealt with at a later time.

The component class contains two strings; the component type, name and the component identifier, id. There are two vectors of ports relating to the decision table and operational mode table columns. Each port of the component (In, Out) contains values and these are captured within the port class. Lastly, the component class also includes the full decision table and, where applicable, an operational mode table as it is found within the file (excluding the port names).

Each port class object created holds the name of the port, for example, In1, and the direction of the port, for example In. This is the only information that is captured from the decision and operational mode table files. Other information held within the port class is added at a later time.
Figure 6.7: Class representation of the component class and its sub-classes
6.3. Software Structure

Decision and Operational Mode Tables

Each file with a file extension, .dt or .omt, would be identified as a component decision table or operational mode table, respectively. Each file would be subject to the algorithm seen in Figures 6.8 and 6.9. This algorithm is used to create a new component class object and then store the port information and the table details. Once this is complete, the new component class is added to the component library class, compLib.

6.3.1.2 Storage of the system structure

The information contained within the system topology file (file extension .ss), gives the overall structure of the system. The information describes links between one component’s input port and another component’s output port. It is important that all component decision and operational mode table files are listed before the system structure file within the project files, as the component decision and operational mode table files describe the component type. The system structure then relies on knowledge of these components and their ports when describing instances of these components. Therefore if a component type is listed within the system structure file that was not previously listed in the component decision and operational mode files then the software identifies that the file is missing and issues an error to the user.

If a component is dependent on the operational mode of another component, then this is presented in the decision tables and the system structure files. Within the decision tables, the component that is dependent has a specific input port that represents the other component’s mode of operation. There is a link for this input port in the system structure file. For the component that has multiple modes of operation, a mode output must be provided to link to the dependent component. An example of this is seen within the pressure tank system from Chapter 5. The component tank, \( T \), is dependent on the operational mode of the component valve, \( V \). Therefore the input port, \( in2 \) of the tank component is linked to the mode, \( mode3 \), of the valve component.

The process for abstracting and storing the information held within the system structure file can be seen in Figures 6.10 - 6.12.

6.3.1.3 Storage of Mission Description

Within the .ptt file there should be only one phase transition table describing the mission. As the information held within this file is parsed, the information is stored in a new phaseTable class object. There are only two variables within the phaseTable class: the name of the mission and the phase transition table contents. The phaseRow class holds a row of the table: time (if present), from phase, to phase and condition. The condition variable is the pttContents class which holds the component information; a link (referred
Figure 6.8: First section of the algorithm for parsing and storing information within a .dt or .omt file
Figure 6.9: Second section of the algorithm for parsing and storing information within a .dt or .omt file
Figure 6.10: First section of the algorithm for parsing and storing information within a .ss file
Figure 6.11: Second section of the algorithm for parsing and storing information within a .ss file
Obtain a link (pointer) to the component class instance

Obtain a link (pointer) to the port class instance

Obtain a link (pointer) to the link class instance

Is the link class instance already connected?

Set the link type (the port values)

Are the number of links associated to the link class instance?

Increase the number of connections to the link class instance

Add the link name to the port class instance

END

END

END

Yes

No

Yes

No

Yes

No

Issue error to user: The link types are not the same

Issue error to user: The link to this component cannot be made as the link has too many connections to create a new link

Figure 6.12: Third section of the algorithm for parsing and storing information within a .ss file
to as pointer) to the component, the component’s port and value of the port.

The process for abstracting and storing this information can be seen in Figures 6.13 - 6.15.

6.3.1.4 Storage of Simulation Information

The simulation file contains the information regarding component failure data and initial component mode for those with multiple modes. It also contains the initiating component: the component identifier, the port name and the value of that port. For example, $OP \ out1 = CL$.

Component failure data

As each set of component failure data in the list is parsed into the software, the following steps are completed:

1. The component instance is located within the $systemTopologies$ variable of the $topSystem$ class.

2. The state or mode used in the failure data is checked against the component instance to ensure the state or mode is present in the component.

3. A new instance of the class $failRepair$ is created. The $failRepair$ class stores information about the component’s failure data.

4. The failure distribution information is then assigned to the class.

5. The class calls a function to calculate the first time to failure, based on the failure data provided.

6. This $failRepair$ class instance is added to the component instance held within the variable $systemTopologies$.

Initial component modes

The initial component modes in the file are parsed and the system handles them in the following manner:

1. The component instance is located in the $systemTopologies$ variable of the $topSystem$ class.

2. The mode used in the initial mode data is checked against the component instance to ensure the mode is present in the component.

3. The port class object in the component that represents the component mode is updated with the specified value.
Figure 6.13: First section of the algorithm for parsing and storing information within a .ptt file
Figure 6.14: Second section of the algorithm for parsing and storing information within a .ptt file
Figure 6.15: Third section of the algorithm for parsing and storing information within a .ptt file
Initial component

The initial component information is stored directly into a variable held within the topSystem class, ‘initial initialComp’, where ‘initial’ is a class with three string variables which store the component identifier, the port name and the port value.

When parsing the file, the software checks the topology class instance to ensure that a component with that identifier, port name and port value exists.

6.3.1.5 Circuit List Generation

After all the files have been entered into the software, a list of circuits in the system is generated. This section describes the recursive function that is used to generate the list of circuits for any given system, as seen in Figure 6.16.

The software identifies the first component within the topology that has wire types, as discussed in 6.2.5, for one of its output ports. Once identified, the software searches for the components that are connected to each of the component’s ports that have a wire link type. Wherever a circuit branches (e.g. at a junction), the same function is called recursively on each branch. At the end of a branch (having returned to the first component in the top-level circuit), a completed circuit is returned. Figure 6.17 demonstrates the building of the circuit lists using the pressure tank system from the previous chapter. The figure shows how, starting from a single component (in this case the power supply PS1), circuits are discovered via a number of branches. Only when the software reaches the original component is a circuit complete and stored by the software.

Calling this small function recursively is an elegant and effective method of mapping all the electrical circuits in a system topology.

6.3.2 Building the Petri Net Model

6.3.2.1 Introduction

Once the system, phase and simulation information has been added, the model can be generated. The model building section of the software uses the component tables and phase table to construct the Petri net model. The model can be split into four distinct parts:

1. Component Petri Net
2. System Petri Net
3. Circuit Petri Net
4. Phase Petri Net
Figure 6.16: Process for the generation of the circuit lists
Each of these are built individually and in the order given above. They are built in this order due to the connections that exist between the different Petri net sections. Each of the following sections discusses the construction of each of the above Petri net types and how each connects to the previous Petri net type.

This section of the software introduces a number of new classes that relate directly to Petri nets. Petri nets comprise of place, transition and arc classes. All Petri nets generated in the software are stored within a single masterPetriNet class instance within the topSystem class.

### 6.3.2.2 Component Petri Net

The component Petri nets are the first to be built as these are the building blocks for all other Petri nets. All component information was added to the software via the .dt, .omt and .sim files. The .dt and .omt files contain the information relating to how the component works under different conditions, and the .sim file contains the information relating to the failure data and initial modes (for components with multiple modes).

This process follows the construction procedure established in section 4.3.1.1. Starting with the information contained within the component class objects, the software moves through each port of the component. In each port, a port name, direction and link type exist. For each link type value, a new place class object is created. For each of the places created, an identifier is created in the following format:

\[
\text{component.componentType.componentID.portName.value}
\]

For example, a place representing a value of \(C\) in an input port \(in1\) of a power supply with unique identifier \(PS1\) would be written as \(\text{component.power_supply.PS1.in1.C}\). This
unique identifier is necessary when making connections between places and transitions within the component Petri net or between other Petri nets.

Each transition created for the component Petri net is based on a row of either a decision table or operational mode table. This distinction is reflected in the identifier used for each of the transitions created:

\[
\text{component.componentType.componentID.tableType.rowNumber}
\]

For a transition representing the second row of a DT for a power supply with the unique identifier \textit{PS1}, the transition identifier would be \textit{component.power_supply.PS1.dt.2}.

Using the information stored about the DT and OMT (if applicable) for each component, arcs are created for each link. The arc class includes a place identifier, a transition identifier and the arc type. Possible arc types are shown in Table 6.1. The software determines the arc type to use according to the procedure discussed in section 4.3.1.1, items 4 and 5.

<table>
<thead>
<tr>
<th>Software Arc Type</th>
<th>Description</th>
<th>Arc Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Single-headed arc</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Double-headed arc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Single-headed inhibit arc</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Inhibit arc</td>
<td></td>
</tr>
</tbody>
</table>

### 6.3.2.3 System Petri Net

The system begins by generating a component instance for every component in the system. The name/type of the component and the component identifier are added to the component Petri net object. If the component has multiple modes of operation then the starting mode is represented by a token in that mode’s place in the component Petri net object instance. There is one last part of the component Petri net that is not built initially with the rest of the Petri net: the working-to-failed relationship. The information stored in the port representing the state of the component is located and passed to the component Petri net to build this information into the Petri net. The process for this can be seen below:

1. Locate the place representing the working state in the component Petri net.
2. If the state port was located within the decision table ports, then for each \textit{failRepair} object associated with the port, create the following:

(a) Create a transition object instance.

(b) Set the time to transition equal to that listed within the \textit{failRepair} object instance.

(c) Create the identifier of the transition which follows;

\[ \text{component.componentName.componentID.timed.failTime} \]

(d) Set the identifier to the transition.

(e) Add the transition to the component Petri net.

(f) Create an arc object instance to show the connection between the working place and the newly created transition.

(g) Locate the failed place of the component

(h) Create a new arc object to represent the connection from the timed transition to the failed place.

3. If the state port was located within the operational mode table then the following steps should be taken:

(a) If the component’s modes fail at the same rate complete the following:

i. Create a transition object instance.

ii. Set the time to transition equal to that listed within the \textit{failRepair} object instance.

iii. Create the identifier of the transition as follows;

\[ \text{component.componentName.componentID.timed.failTime} \]

iv. Set the identifier to the transition.

v. Add the transition to the component Petri net.

vi. Create a new place instance to represent the failure of the component.

vii. Add this new place to the component Petri net.

viii. Create an arc object instance to show the connection between the working place and the newly created transition.

ix. Create an arc object instance to show the connection between the timed transition and the newly created transition.

x. Create a new immediate transition object for each failure mode place.

xi. Create an arc instance to represent the connection between the failure place and the new transition.
xii. Create an arc instance to represent the connection between the component mode place and the transition.

xiii. Create an arc instance to represent the connection between the transition and the failure mode place.

(b) If the component’s modes do not fail at the same rate then complete the following:

i. Create a transition object instance for each failure mode.

ii. Set the time to transition equal to that listed within the failRepair object instance for each failure mode.

iii. Create an arc instance for each failure mode to represent the connection between the working place and the timed transition.

iv. Create an arc instance for each mode to represent the connection between the component mode place and the transition.

v. Create an arc instance for each mode to represent the connection between the timed transition and the failure mode place.

The rest of the SPN is created using the system topology information provided by the user to generate the connections between the components. This consists of locating the output place of one component and an input place of the connected component and merging these places so that only one place remains. The algorithm of the software follows the construction procedure seen in section 4.3.3.1.

6.3.2.4 Circuit Petri Net

After the circuits are detected as described in section 6.3.1.5, the CiPNs can be generated. Once created, each CiPN is stored within an instance of a circuitPN class. This contains a vector that holds all CiPN instances within the masterPetriNet class. The algorithm for generating these Petri nets follows the construction procedure outlined in section 4.3.2.1.

The software generates a number of places to represent current and no current within a given circuit. These use the following place identifier format:

\[ \text{circuit\.circuit\_number\.wireType} \]

The places are defined as the circuit control places have the following format:

\[ \text{controlPlace\_circuit\.circuit\_number\.wireType} \]

The transition identifiers created as part of the CiPN show the relationship between the wire type and the component. For example, the transition that would link a component’s mode to a circuit’s no current place would have the following transition identifier:

\[ \text{circuit\.circuit\_number\.component\.componentID\_transition\.mode} \]
6.3. Software Structure 177

6.3.2.5 Phase Petri Net

The algorithm of the software for generating the PPN follows the construction procedure seen in section 4.3.4.1.

The main phase places are the places that the system should move through if the mission completes successfully without any failures. The main phase places are identified within the software by the place identifier:

\[ \text{phase.phase\_Number.start\_time-} \rightarrow \text{end\_time} \]

If the place is used to represent a specific moment in time then the place identifier uses the following format:

\[ \text{phase.time.time\_value} \]

Other phase places, such as failure places, generated from the phase transition table are identified by the following format:

\[ \text{phase.phase\_Number} \]

There are a number of different types of transitions created. The timed transitions created as part of the model creation from the phase transition file (.ptt file extension) are given the following identifier:

\[ \text{phase.timed.time\_value} \]

If the transition has a delay of \( \delta \) then the transition is given the following identifier:

\[ \text{phase.timed.delta.row\_number} \]

Where \( \text{row\_number} \) relates to the row number within the phase transition table. Other transitions created from the phase transition table use the following format:

\[ \text{phase.table.row\_number} \]

Time Dependent Components

During this stage of construction the software moves through each decision table searching for any component that is dependent on time. If a component is located then the software assess whether there is already a suitable PPN place that represents this specific moment in time or a given time frame. If there is no place suitable then the software creates a new place to represent this time frame. The time frame is checked to see that it can be related to a PPN place first. Components should only change behaviour according to times related to a phase within the mission. If a new place is required the place identifier follows the following format:
phase.dt.start_time->end_time

If a new place is required then a new transition is also necessary. The transition connects between the place that represents the start time of the time frame and the new place created above. The new place then connects to the transition that would then enter a place representing the end time. This place would then connect to the relevant immediate transition created during the CPN model creation. The transition generated here would have the following identifier format:

phase.transition.start_time->end_time

If the inequality expressing the time frame is inclusive (i.e. greater than or equal to, less than or equal to) then this is also demonstrated within the identifier by use of an equals sign between start_time and − > or between − > and end_time.

When the PPN is generated it is stored within the phasePN class instance that is held within the masterPetriNet class instance. All final Petri nets are stored here until they are required for the simulation which is described in the next section.

6.3.3 Simulating the Petri Net Model

The Petri net simulator created as part of the software is the most complex part of the entire software. The simulator was designed to handle the type of model discussed in section 4.3. The simulator has one particular feature that has not been identified within any other Petri net simulator: the ability to wipe clean specific tokens within a Petri net model. It is very important that this can occur when handling the model generated. The details of this feature and its purpose are described within this section.

This section discusses the main class associated with the simulation of the model, simulation. This class is described in detail from the initial preliminary simulation of the model to the main simulation and the main functions used to simulate each transition in the Petri net. The main algorithm used to simulate the whole model is also discussed. This algorithm assesses the system at different stages to find the moment at which the simulation has run its course.

Before the simulation functions are executed, the Petri nets that are stored within the masterPetriNet class instance are transferred to the simulation class instance. The Petri nets are transferred as separate vectors of places, transitions and arcs. Each of the places and transitions are assigned integer identifiers as string identifiers take considerably longer to validate during a simulation run. These identifiers are then also stored within the arcs for reference. At this point the Petri nets are no longer identifiable as discrete from one another, but form a single, large Petri net. The matrices required to simulate the transitions firing during a simulation are then created. The method of marking transformation is
6.3. Software Structure

described in section 1.4.3.2. The software generates the incidence matrix, $A^T$, using the places and the arcs to determine the values. This is done automatically the user has no further input at this point. Once each of these matrices are created the simulation process can commence.

6.3.3.1 Simulation Algorithm

The user can initiate a simulation from the main menu of the software. The user specifies the number of simulations to run and the duration of each simulation, which are the only two arguments to the simulation command. This section describes the different algorithms used when a simulation is run. These algorithms are as follows:

1. Before the actual simulation can begin, the input data is processed automatically to locate key elements of the model, separated as follows:

   (a) All simulation end places including the mission success place and failure phases that terminate the mission
   (b) All places that represent component states
   (c) All places that represent component modes
   (d) All places related to the circuit Petri net with IDs containing the key words `controlPlace` and `state`
   (e) All the phase timed transitions
   (f) All places that represent phases including main phases that the system moves through during a mission with no failures
   (g) All other phase places
   (h) All timed transitions (phase timed transitions and component failure timed transitions)
   (i) Times to failure and phase times

2. A preliminary simulation is performed in which all component times to failure are set a large value, i.e. they will not fail. The purpose of this simulation is to validate the model, ensuring that mission success is feasible. The preliminary simulation also analyses components with multiple modes and times to failure. The simulator measures the time spent in each mode to determine which failure mode will occur first. This informs how and when the component fails during the full simulation.

3. All the information required to run the full simulation is now available, including component failure times generated during model construction. The mission involving the first component failure is now simulated. The preliminary simulation has
established that the mission is successful when no components fail; therefore the simulator identifies in which mission the first component fails. For example, with a mission length of 20 hours and a component failing no earlier than 65 hours, the simulator would begin at 60 hours, having assumed three successful missions.

4. After a single mission has been simulated, the function returns a value: the number of tokens located in the mission success place (0 or 1). If the value is 1 then the simulation was successful despite one or more component failures. If the mission end time is still less than the total simulation period, the simulation is set to run again, taking into account the failed components.

5. If the simulation comes back again as a mission success, then this shows that the component that failed is not a component that can cause a mission failure on its own. Further missions are simulated. If no new components fail in a mission, the simulator will skip ahead to the next mission in which a component fails, as it did in the first mission.

6. The simulator continues to run missions until the simulation time (specified by the user) elapses. If at any point a mission is unsuccessful, then the simulation ends early and the data is collected and written to file.

7. When a simulation ends, either in success or failure, new component failure times are generated and inserted into the model for the next simulation.

8. This process is repeated for the number of simulations stipulated by the user.

9. The simulation is complete and the results are written to two files.

6.3.3.2 Simulation of the Model

This section describes the process for executing the timed transitions in the model. To begin, all timed transitions are identified within the model and their positions are stored locally within the simulation class. These times are then sorted in preference of earliest first.

The function that completes this specific set of tasks is the simulateModel function. This manages when different transitions fire.

There are a set activities that must be completed before the main section of the function. These are covered below:

Mission Length

The user is not required to provide a definitive mission length as this information is abstracted from places created as a result of the phase transition table. The main phase
places (not mission failure or success places) created as part of the modelling process described in section 6.3.2.5 are identified. As the identifier for the places includes the time frame that the phase represents, this value is taken and temporarily stored. When all time frames are identified these are sorted, earliest first. From this list, the largest time value can be determined and set as the length of an individual mission.

**Active Time**

A concept of *active time* is used within the software as a method of tracking the *age* of a transition. This is particularly important for components with multiple modes of operation with different failure rates. This active time is determined using a preliminary run of the system and mission entered by the user, as described in section 6.3.3.1, step 2. The active time is used to calculate the true overall time to failure of the component in the specific mode.

**Enabled Timed Transitions**

This is a list of all timed transitions that have been determined as enabled. To identify which timed transitions are enabled, each time value is tested to see if it would be possible to fire this transition. If it is possible then the transition is added to the list of enabled timed transitions. The identifiers of the transitions are stored for reference for when the enabled value is fired.

**Mission Start and End Time**

Before a mission is run, it is necessary to determine the earliest enabled time. This process was discussed briefly in 6.3.3.1, step 3. The phase times are not considered during this process as this is to determine the earliest time to failure. The earliest time to failure is then used to work out when the mission should commence. The preliminary run of the system and mission ensures that the model created is viable. If the system and mission are not viable then the simulation of the preliminary run would return a failure of the mission without a component failing. It is assumed that the mission would be successful if all components are working. By assuming this the software locates the earliest time to failure of a component and re-works the new mission start time. The value is reworked by completing the following:

1. Take the earliest time to failure and divide it by the mission length.
2. Remove any remainder from the value calculated in 1.
3. The start time is then calculated by multiplying the value calculated in 2. by the mission length.
4. The end time is the addition of the start time and the mission length.

After the start time and end time are calculated, these values are tested against the user defined simulation length. If the start or end time is beyond the simulation time then this simulation is determined to be completed successfully. If the start and end times are within the limits then the simulation can begin.

**Unchanging Place Tokens**

During a simulation the system is wiped of tokens, with some exceptions. These unchanging places are the component working and failed places, circuit control places and all places representing phases of the mission. Before the simulation commences, the number of tokens in each is stored within the *simulation* class. This is to track the tokens within these places.

### 6.3.3.3 Simulating Transitions

To simulate the transitions firing, the method of marking transformation as discussed in section 1.4.3.2 is used. This is implemented in the software using a matrix class written specifically for the software. An instance of this class contains the matrix elements themselves and functions to perform matrix operations including addition, multiplication and transposition.

There are two functions that manage the simulation of the transitions: *simulateTransitions* and *createFiringValues*. The process used by the function *simulateTransitions* can be seen in Figure 6.18. Blocks (4) and (9a) mention an intermediate step matrix: this holds the matrix that results from the multiplication of the $A^T$ and $\sum$ matrices.

The function *createFiringValues* determines which transitions are enabled based on the places and arcs connected to each transition. The function completes the following tasks:

1. Identify the places that are inputs to the transition
2. For single- and double-headed arcs: do all the places contain a token?
   a. Yes: Move on to 3.
   b. No: This transition is not enabled. If there are any more transitions to assess return to 1.
3. For inhibit arcs: do any of the places contain a token?
   a. Yes: This transition is not enabled as it is inhibited. If there are any more transitions to assess return to 1.
   b. No: Move on to 4.
4. Identify the places that are outputs from the transition
5. For single-headed inhibit arcs: do any of the places contain a token?
   a. Yes: This transition is not enabled as it is inhibited. Return to 1. for the next transition.
   b. No: Move on to 6.

6. The transition is enabled. Set the value within the $\Sigma$ matrix to 1. If there are any more transitions to assess return to 1.

### 6.3.3.4 Model Simulation Algorithm

Figures 6.19 and 6.20 show the process that works through the simulation of the model. Within Figure 6.19 there is a key which shows a set of blocks. These blocks show the order in which the timed transitions are considered. Each block contains a variable $StartLocation$ which is used within the software as an identifier to keep track of where the simulation is in relation to each block. It is especially important when (6a) occurs as this breaks out of the loop shown in the key to assess the enabled time transitions.

Blocks (3bi) to (3biv) in Figure 6.20 are required when a phase delta transition is dependent on a component’s output value to move between phases. As tokens flow around the model, it cannot be guaranteed that the output place contains a token at the correct time. To aid this, these blocks continuously move through time = delta and time = 0 transitions until the token is present.

Blocks (6ci) and (6d) in Figure 6.19 set the begin time of a newly enabled time transition. When moving through the blocks seen in the key in Figure 6.19, the simulation could break out of the loop before or after the next enabled time transition has fired. If a transition has become enabled before the next enabled time transition has fired then the start time for this component is the last enabled time transition to have fired (Block(6ci)). If there has been no previously fired time transitions then the value is set to the start of the mission. If the transition has become enabled after the next enabled time transition has fired then the start time is set to this value (Block (6d)).

### 6.3.3.5 Simulation Results

The results from each simulation are logged to file and held in memory. This is so that if the simulation were to be stopped part way through by design or accident the results of the simulation are not lost. There are three files generated as part of the results and are listed below:

* Simulation_results.txt
* Supplementary_details.txt
(1) Is this a delta transition and is this not the first iteration?
No

(1a) Call the simulateTransition for time = 0

(2) Set M₀ matrix to the M₀ matrix.

(3) Update the tokens in the places from the M₀ matrix.

(4) Clear the contents of the intermediate step, M₀, and Σ matrices.

(5) Create the Σ matrix.

(6) Are there any enabled timed transitions and is the time not delta?

(6a) Populate the intermediate step matrix with the multiplication of A' and Σ.

(6b) Populate the M₀ matrix with the addition of the intermediate step matrix and M₀.

(6c) Has there been a change in the unchanging places?
Yes

(6d) Has the mission moved into a new phase?
Yes

(6e) Has the software finished assessing this time?

Yes
No

STOP

Figure 6.18: Flow chart of the process of the simulation of a transition
Figure 6.19: Flow chart of the process of the simulation of the model, part 1
Simulate the transition at time associated to the block

Set the token values of the unchanging places

Is the mission length less than the enabled timed transition?

Is the enabled timed transition greater than the simulation length?

Is the enabled timed transition greater than the end time of the current mission?

Simulate the transition at time associated to the block

Add the enabled timed transition value to the fired transition list

Set the token values of the unchanging places

Was the timed transition a phase transition?

Add the timed transition identifier to a list of failed components

Simulate time = 0

Has the simulation come to an end?

Simulate time = delta

Has the simulation come to an end?

Figure 6.20: Flow chart section of the process of the simulation of the model, part 2
• ExcelFileSimResults.txt

The first file has the unreliability data for each set of 50 simulations and the final unreliability data at simulation end. The second file details each simulation run including the phase sequence and the component failures that have occurred during the mission. The third file is a tab-delimited file of the results that can be opened with excel to generate graphs and other data. A time-stamp of when each of the files is created (at the start of the simulation process) is included.

6.3.3.6 Summary

The simulator created as part of this software is complex but necessary to prove the validity of the models generated using the software. This section has covered the main aspects and processes used to generate reliability data for the model completing a mission or multiple consecutive missions. In order to prove whether the results obtained from the simulation are within acceptable tolerances, an analytical solution must be calculated. The following section discusses the modelling method used to generate the analytical values.

6.4 Testing and Validation

In order to validate the modelling method, the system described in Chapter 5 was used. Files representing the decision and operational mode tables, seen in Tables 5.1 - 5.15, were created. The topology diagram seen in Figure 5.2 was converted into the system structure file format as discussed in Section 6.2.2. The simulation file was populated with the information displayed in Table 6.2 and the following initial conditions were used:

• Push switch, S1 : mode = open

• Toggle switch, S2 : mode = closed

• Relay contact, CR : mode = open

• Timer relay contact, CT : mode = open

• Valve, V : mode = closed

Each component follows an exponential distribution. To validate the simulation results, a set of analytical values were calculated to compare against those generated from the software. To calculate the unreliability of the system over the mission, the reliability modelling method of phased fault trees as discussed in section 2.2.1 was used.
Table 6.2: Pressure tank system component failure data

<table>
<thead>
<tr>
<th>Component identifier</th>
<th>Failure Mode</th>
<th>Failure Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>F_closed</td>
<td>0.1</td>
</tr>
<tr>
<td>S1</td>
<td>F_open</td>
<td>0.1</td>
</tr>
<tr>
<td>S2</td>
<td>F_closed</td>
<td>0.8698</td>
</tr>
<tr>
<td>S2</td>
<td>F_open</td>
<td>0.001</td>
</tr>
<tr>
<td>PS1</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>PS2</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>CT</td>
<td>F_closed</td>
<td>0.1</td>
</tr>
<tr>
<td>CT</td>
<td>F_open</td>
<td>0.1</td>
</tr>
<tr>
<td>CR</td>
<td>F_closed</td>
<td>0.00023</td>
</tr>
<tr>
<td>CR</td>
<td>F_open</td>
<td>0.00023</td>
</tr>
<tr>
<td>TIM</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>R</td>
<td>F</td>
<td>0.1</td>
</tr>
<tr>
<td>M</td>
<td>F</td>
<td>0.001</td>
</tr>
<tr>
<td>FS</td>
<td>F</td>
<td>0.01</td>
</tr>
<tr>
<td>P</td>
<td>F</td>
<td>0.1</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>0.0001</td>
</tr>
<tr>
<td>V</td>
<td>F_closed</td>
<td>0.03</td>
</tr>
<tr>
<td>V</td>
<td>F_open</td>
<td>0.03</td>
</tr>
<tr>
<td>PG</td>
<td>F_LOW</td>
<td>0.01</td>
</tr>
<tr>
<td>PG</td>
<td>F_HIGH</td>
<td>0.01</td>
</tr>
<tr>
<td>PG</td>
<td>F_VHIGH</td>
<td>0.01</td>
</tr>
<tr>
<td>OP</td>
<td>F</td>
<td>0.1</td>
</tr>
</tbody>
</table>

6.4.1 Validation using Phase Fault Trees

To validate that the software provides results accurate to within a tolerance of ± 5%, the software was set to run for a single mission. Using the relationship between failure rate and mean time to failure ($\mu = 1/\lambda$), the failure rates in Table 6.2 were converted into their equivalent mean times to failure. These were entered into the .sim file.

The phase fault trees for this mission can be seen in Figures 6.21 - 6.24. Each phase fault tree was evaluated using the data given in Table 6.2 and using the procedure seen in section 2.2.1.3 by La Band (2005). The subscript numbers within the basic events demonstrate when these basic events can occur i.e. in which phases these basic events can occur which would contribute to the failure in the current phase. This generally affects components with multiple modes of operation, as the time in which the component fails dictates the mode in which the component fails.

The phase unreliability was calculated using the minimal cut set upper bound approximation described in Section 1.4.1.1. This was applied to each of the phase fault trees and can be seen in equations 6.4.1, 6.4.2, 6.4.4 and 6.4.6 to represent the unreliability of phase 1, 2, 3 and 4, respectively.
6.4. Testing and Validation

Figure 6.21: Pressure tank system phase 1 fault tree

Figure 6.22: Pressure tank system phase 2 fault tree
Chapter 6. Automated Reliability Modelling

Figure 6.23: Pressure tank system phase 3 fault tree

Figure 6.24: Pressure tank system phase 4 fault tree
6.4. Testing and Validation

\[ Q(\text{Phase}1) = 1 - (1 - P(S1OP1))(1 - P(PS1))(1 - P(TCOP1))(1 - P(TIM1)) \] \hspace{1cm} (6.4.1)

\[ Q(\text{Phase}2) = 1 - (1 - P(X \cdot R_{12}))(1 - P(X \cdot RCOP1))(1 - P(X \cdot M_{12})) \]
\[ (1 - P(X \cdot FS_{12}))(1 - P(X \cdot PS_{212}))(1 - P(X \cdot T_{12}))(1 - P(X \cdot P_{12})) \]
\[ (1 - P(S1OP1 \cdot PS1 \cdot TCOP1 \cdot TIM1)) \]
\[ (1 - P(S1OP1 \cdot TCOP1 \cdot TIM1 \cdot PS1)) \] \hspace{1cm} (6.4.2)

Where \( X \) is defined by the equation 6.4.3.

\[ X = S1OP1 \cdot PS1 \cdot TCOP1 \cdot TIM1 \] \hspace{1cm} (6.4.3)

\[ Q(\text{Phase}3) = 1 - (1 - P(Y \cdot V_{CL_{13}}))(1 - P(Y \cdot OP_{13}))(1 - P(Y \cdot PG_{L_{13}})) \] \hspace{1cm} (6.4.4)

Where \( Y \) is defined by the equation 6.4.5.

\[ Y = S1OP1 \cdot PS1 \cdot TCOP1 \cdot TIM1 \cdot RCOP1 \cdot PS2 \cdot R_{12} \cdot M_{12} \cdot FS_{12} \cdot P_{12} \cdot T_{12} \cdot RC_{OP1} \cdot PS2_{12} \] \hspace{1cm} (6.4.5)

\[ Q(\text{Phase}4) = 1 - (1 - P(Z \cdot RC_{CL_{12}}))(1 - P(Z \cdot S1_{CL1}))(1 - P(Z \cdot TC_{CL_{12}})) \] \hspace{1cm} (6.4.6)

Where \( Z \) is defined by the equation 6.4.7.

\[ Z = S1OP1 \cdot PS1 \cdot TCOP1 \cdot TIM1 \cdot RC_{OP1} \cdot PS2_{12} \cdot V_{CL_{13}} \cdot OP_{13} \cdot PG_{L_{13}} \] \hspace{1cm} (6.4.7)

Each of the above equations were calculated and the results obtained can be seen in Table 6.3. The next section details the results obtained from the software with a comparison to the values obtained using the above equations.

6.4.1.1 Single Mission Simulation Results

The software completed 10,000 simulations of a single mission of the pressure tank system. The final simulation results obtained can be seen in Table 6.3. Figures 6.25, 6.26 and
Table 6.3: Pressure tank system simulation results from 10,000 simulations

<table>
<thead>
<tr>
<th>Phase Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytical</td>
<td>0</td>
<td>0.1927</td>
<td>0.1061</td>
<td>0.0670</td>
<td>0.3658</td>
</tr>
<tr>
<td>Simulation</td>
<td>0</td>
<td>0.1990</td>
<td>0.1080</td>
<td>0.0645</td>
<td>0.3715</td>
</tr>
<tr>
<td>Difference(%)</td>
<td>0</td>
<td>3.25</td>
<td>1.80</td>
<td>3.74</td>
<td>1.56</td>
</tr>
</tbody>
</table>

Figure 6.25: Phase 2 simulation convergence results

6.27 show the phase unreliability for phases 2, 3 and 4 respectively. These show how the simulation results begin to converge on the analytical values calculated in section 6.4.1.

6.4.1.2 Analysis

Convergence Study

To validate the results a convergence study was carried out for each individual phase of the mission and the overall unreliability of the mission. The first phase of the mission, a discrete phase, was completed successfully during each mission run during the simulation. The further three phases of the mission (2, 3 and 4) began to converge at approximately 2,100 simulations. This is a convergence within a ± 5% tolerance. The mission unreliability can be seen to converge at approximately 1,250 simulations within a ± 5% tolerance.

To show this convergence the values between 2,000 simulations and 10,000 simulations
Figure 6.26: Phase 3 simulation convergence results

Figure 6.27: Phase 4 simulation convergence results
were taken and displayed in Figure 6.29. This shows the values staying within the tolerances of ±5%.

The simulation results shown for the convergence study and the simulation of multiple missions, discussed in the next section, were published in Stockwell and Dunnett (2013).

### 6.4.1.3 Simulation of Multiple Missions

The previous sections discussed using the software for calculating single mission unreliability. This section takes this one step further and discusses using the software for calculating the unreliability of multiple continuous missions. For this type of simulation it is assumed that the system does not shutdown, but as one mission ends the next begins. For this study the failure data in Table 6.2 was divided by $10^3$. This was to enable components to last multiple missions. As the reliability and the availability of a non-repairable system are equal as discussed in section 1.3.4.1. Figure 6.30 shows the unavailability of each phase of the system over 5,000 consecutive missions. Figure 6.31 shows the unavailability of the system over 5,000 consecutive missions. As expected, Figure 6.31 shows that increasing the demand on the system reduces the probability of mission success. At approximately 3,000 missions the unavailability of the system is very close to zero. The curve of the data follows that of an exponential function. This is unsurprising given that this system is a
Figure 6.29: Section of the mission unreliability graph for a non-repairable pressure tank system
non-repairable system and that each component within the system follows an exponential failure distribution.

6.5 Summary

This chapter has detailed the three main aspects of the software created: file input, model creation and simulation. Each of these has been discussed in detail to show the complexity of the software generated. It has described a number of the major classes and functions that make up the software. Flow charts of the procedures that the software completes in order to achieve specific goals have been shown.

The files that the user must create are simple and require no specialist knowledge of risk and reliability techniques. The tables that describe the system topology, mission and component behaviours are intuitive for design engineers and are created as simple plain text files. There are defined instructions for the user to generate each file type and how to change any predefined values and libraries of components can be generated and stored for future use. To generate the model and perform a simulation, simple commands with minimal arguments are entered into the main menu of the software.

The model building section detailed how the software uses the novel techniques discussed in chapter 4 to generate each of the four Petri net models: CPN, SPN, CiPN.
6.5. Summary

and PPN from the user input files. The different Petri nets are assembled by the software to form a single model of the overall system.

The software handles items such as automatically generating circuit lists without any further aid from the user. This means the model construction process is repeatable between different systems and different users. As long as the user defines the system correctly within the component tables and the system topology the software will identify each circuit within a system.

The implementation of a new simulator, written specifically for the generated models, was discussed in detail. The simulator can perform a number of simulations specified by the user to calculate the unreliability of the system across a single mission or multiple successive missions, making the software suitable for a wider range of systems. The simulator outputs unreliability data as well as more detailed data on phase failures and component failures.

The results validate the novel model generation technique and show great potential for the software developed. The software proved that it can calculate mission unreliability that agrees with an analytical solution to within a tolerance of ±5% for individual phases and for the overall mission.
Chapter 7

Modelling of Repairable Systems

Contents

7.1 Introduction .............................. 200
7.2 Preventative Maintenance .................. 201
  7.2.1 File Input ............................. 201
  7.2.2 System Storage ......................... 202
  7.2.3 Construction Procedure ................. 203
7.3 Corrective Maintenance .................... 206
  7.3.1 File Input ............................. 207
  7.3.2 System Storage ......................... 207
  7.3.3 Construction Procedure ................. 208
7.4 Standby Systems ......................... 212
  7.4.1 File Input ............................. 212
  7.4.2 System Storage ......................... 215
  7.4.3 Construction Procedure ................. 217
  7.4.4 Cold Standby ......................... 218
  7.4.5 Warm Standby ......................... 219
  7.4.6 Hot Standby ............................ 219
7.5 Voting Systems ............................ 221
  7.5.1 File Input ............................. 221
  7.5.2 System Storage ......................... 222
  7.5.3 Construction Procedure ................. 222
7.6 Mission Abort ............................. 225
  7.6.1 File Input ............................. 225
  7.6.2 System Storage ......................... 225
  7.6.3 Construction Procedure ................. 225
7.7 Simulating a Repairable System ............ 229
  7.7.1 Simulation Algorithm .................... 229
  7.7.2 Simulation of the model .................. 230
  7.7.3 Simulating Transitions .................. 230
7.1 Introduction

The procedure given in Chapter 4 is built upon in this chapter to cater for systems that can be repaired during a mission. The software has been designed so that any system can have both repairable and non-repairable components. For the purposes of demonstrating the repairable capabilities, all components within this chapter are assumed to be repairable. The standard Petri net to represent failed to repair can be seen in Schneeweiss (1999), an example of which can be seen in Figure 7.1. Figure 7.1 shows the original working to failed states, seen previously in Figure 4.8a, with the addition of a new place to represent the under repair status of a component. An immediate transition is placed between the failed state place and the under repair state place. This shows that once the component enters a failed state it is automatically assumed to be under repair. Between the under repair state place and the working state place a timed transition, $t_R$, is created. This represents the time the component requires to be repaired to bring it back to a fully working state.

![Figure 7.1: Petri net showing the transition between working, failed and under repair states of a component](image)

This chapter focuses on the repairable nature of systems by looking at different methods of maintenance and the use of standby systems and voting systems as methods of redundancy. The maintenance methods covered in this work include preventative and corrective maintenance. Redundancy has been investigated through standby systems: cold,
7.2 Preventative Maintenance

Preventative maintenance or scheduled maintenance as discussed in section 1.3.4.2 has been considered in the software. It has been considered both on a system-wide level and component-by-component basis. The Petri net that represents the preventative maintenance for any given component can be seen in Figure 7.2. This is based on a version of the Petri net found in Schneeweiss (1999). The Up and Down places represent that the component is under a preventative maintenance cycle and not, respectively. Additional inhibit arcs between the failed place, under repair place and the transition representing the maintenance cycle coming to an end were created. This is to ensure that the preventative maintenance cycle completes the repair before returning to the Down place.

This section describes the user expressions required within the input files and the manner in which the information is stored and used to build the Petri net model seen in Figure 7.2.

7.2.1 File Input

All maintenance plans are expressed within the simulation file (.sim). This includes system-wide and component-based maintenance plans. These are written within a MAINTENANCE header in the file. The preventative maintenance cycle has been modelled for individual components by the user stipulating the following:

\[ \text{preventative componentID (time\_between\_maintenance\_cycles);} \]

or it can be stipulated for the system by using the following expression:
An example of the entry for a preventative maintenance cycle for both a system-wide event and a component-specific plan can be seen in Figure 7.3.

The time between cycles is assumed to be divisible by the mission length as it is assumed that preventative maintenance cycles are not completed within a running mission, but at the end of a mission.

### 7.2.2 System Storage

There are two classes that are used when storing this information from the simulation file: `planDetails` and `preventative`. The `planDetails` class can be seen in Figure 7.4. This class forms the basic information for each maintenance plan. This is used to stipulate whether the plan is component based, in which case the variable `componentId` would be populated. If the plan is system wide then the `systemStatus` variable would be set to `true`. The `preventative` class inherits the attributes within the `planDetails` class. This is also true of the `corrective` class to be discussed in the next section. The `preventative` class can be seen in Figure 7.5; this class has one variable `duration` which is the time between maintenance cycles.

```c++
class preventative :
    public planDetails

    - double duration;
```

Figure 7.5: preventative class view
7.2. Preventative Maintenance

All maintenance plans are stored within a maintenance class instance which holds a vector of each maintenance item and any standby and voting system plans as seen in Figure 7.6. The maintenance class instance is a variable added to the topSystem class as part of the repairable work shown here.

The procedure for taking the file and storing the information for preventative maintenance cycles has been detailed in Figure 7.7. The Petri net construction process based on this information is considered in the next section.

7.2.3 Construction Procedure

The construction procedure detailed here takes the information stored within the preventative class instances held within the maintenance class instance to generate the Petri net model.

1. Create a new place to represent the simulation is under maintenance: simulation.maintenance

2. Consider each preventative plan: Is this a system wide plan?

   (a) Yes

      i. Create a place to represent the system undertaking preventative maintenance: component.system.wide.preventative.up

      ii. Create a place to represent the system not undertaking preventative maintenance: component.system.wide.preventative.down. Set the place to have a single token.

      iii. Create a timed transition to represent the time between maintenance cycles. This transition has the following identifier: component.system.wide.timed_p.time_between_cycles

      iv. Create the following arcs:

         A. Create a single-headed arc between the place created in 2(a)ii and the transition to represent time between cycles created in 2(a)iii.
Figure 7.7: Flow chart showing the steps to taking the file input and storing the preventative maintenance information
B. Create a single-headed arc between the timed transition created in 2(a)iii and the place representing the undertaking of maintenance created in 2(a)i.

v. Create an immediate transition. This immediate transition allows the transition between the maintenance up place to the maintenance down place. The identifier for the transition is as follows: component.system.wide.immediate_p.upToDown.

vi. Create the following arcs:
   A. Create a single-headed arc from the under maintenance place created in 2(a)i to the immediate transition created in 2(a)v.
   B. Create a single-headed arc from the immediate transition created in 2(a)v to the not under maintenance place created in 2(a)ii.
   C. Create an inhibit arc from each component failed places to the immediate transition created in 2(a)v.

vii. Identify each component in the system that is repairable and not already maintained through a corrective maintenance plan and complete the following:
   A. Create an inhibit arc from each component failed place to the immediate transition created in 2(a)v.
   B. Create an inhibit arc from the component’s under repair place to the immediate transition created in 2(a)v.
   C. A double-headed arc from the under maintenance place created in 2(a)i to each of the immediate transitions that link a component’s failed place and under repair place.

(b) No

i. Retrieve the component associated to the plan.

ii. Create a place to represent that the component is undertaking maintenance: component.componentType.componentID.preventative.up

iii. Create a place to represent that the component is not undertaking maintenance: component.componentType.componentID.preventative.down.
   This place is set to have one token, as all components are assumed to be working from time zero.

iv. Create a timed transition to represent the time between maintenance cycles. The identifier for this transition is component.componentType.componentID.timed_p.time_between_cycles

v. Create the following arcs:
A. Create a single-headed arc between the place created in 2(b)iii and the transition to represent time between cycles created in 2(b)iv.

B. Create a single-headed arc between the timed transition created in 2(b)iv and the place representing the undertaking of maintenance created in 2(b)ii.

vii. Create an immediate transition. This immediate transition allows the transition between the maintenance up place to the maintenance down place. The identifier for the transition is as follows:

\[ \text{component.componentType.componentID.immediate\_p.\_upTo\_Down} \]

vii. Create the following arcs:

A. Create a single-headed arc from the under maintenance place created in 2(b)ii to the immediate transition created in 2(b)vi.

B. Create a single-headed arc from the immediate transition created in 2(b)vi to the not under maintenance place created in 2(b)iii.

C. Create an inhibit arc from each component failed places to the immediate transition created in 2(b)vi.

D. Create an inhibit arc from the component’s under repair place to the immediate transition created in 2(b)vi.

E. A double-headed arc from the under maintenance place created in 2(b)ii to each of the immediate transitions that link a component’s failed place and under repair place.

An example of a single component has been given in Figure 7.2. An example of a system wide preventative maintenance plan has been given in Figure 7.8. This shows three components within a system that are all repairable and are not covered by another maintenance plan.

The next maintenance plan type to consider is corrective maintenance which is covered in the next section.

### 7.3 Corrective Maintenance

Corrective maintenance or unscheduled maintenance as discussed in section 1.3.4.2 has also been considered in the software. As with preventative maintenance, corrective maintenance can be considered on a component-by-component basis or as a system-wide event. Both will be discussed here. The Petri net structures used in this section are based on Schneeweiss (1999).
7.3. Corrective Maintenance

Figure 7.8: Flow chart showing the steps to taking the file input and storing the preventative maintenance information

7.3.1 File Input

The declarations for each type of corrective maintenance plan is given within the MAINTENANCE heading within the simulation file. The declaration for a component is as follows:

`corrective componentID(maintenance_engineer_identifier);`

and the declaration for a system wide corrective maintenance plan is as follows:

`system corrective (number_of_personnel);`

The component corrective maintenance declaration requires an identifier to show which maintenance engineer completes maintenance for the component identified. This is to cater for the possibility that a single maintenance engineer would cover the maintenance for multiple components within a system. The system-wide corrective maintenance plan requires the number of maintenance personnel that are available to carry out the maintenance of the system.

7.3.2 System Storage

Each individual corrective maintenance plan is stored within a corrective class instance. The class view can be seen in Figure 7.9. The corrective class inherits the attributes of the planDetails class. The component that the plan applies to (or the system-wide event indicator) is stored within the attributes of the planDetails class. Within the corrective
class corrective : public planDetails
    -vector<string> personnel;
    -unsigned int noOfPersonnel;

Figure 7.9: corrective class view

class, a list of maintenance engineer identifiers (as discussed in 7.3.1) is stored, relating
to individual component corrective maintenance. Also stored within the corrective class is
the number of maintenance personnel available for system-wide maintenance. The set of
corrective classes are stored within the maintenance class instance as seen in Figure 7.6.
The process for storing this information can be seen in Figure 7.10.

7.3.3 Construction Procedure

The construction procedure for a corrective maintenance plan has been described below:

1. Identify all engineer identifiers from each corrective maintenance plan

2. For each identified in 1 create a place to represent that engineer is available to
   complete maintenance. The identifier would use the following format: component.corr.eng.engID.up

3. For each place identified in 1, create a place to represent that the engineer
   has completed maintenance. The identifier would follow the following format:
   component.corr.eng.engID.down

4. Is this a system-wide event?
   
   (a) Yes:

      i. Create a place to represent the engineers available. The place identifier
         would follow the following format:
         system.corr.eng.plan_number.number_of_engineers

      ii. Set the number of tokens in the place created in 4(a)i to the number of
          engineers stated in the file ('number_of_engineers')

      iii. Identify the repairable components in the system (components with an
           under repair place) and complete the following:

           A. Retrieve the transitions that are linked between the failed places and
              under repair place

           B. Create a single-headed arc between the place created in 4(a)i and the
              transitions identified in 4(a)iiiA
Figure 7.10: Flow chart for the process of taking in the file and interpreting the corrective maintenance plan
C. Retrieve the time to repair transition of the component
D. Create a single-headed arc between the time to repair transition and
   the place created in 4(a)i
E. Retrieve the failed places for the current component
F. For every other repairable component in the topology listed after the
   current component complete the following:
   - Locate the immediate transitions connected between the failed places
     and the repair place for the components identified
   - Create an inhibit arc between the failed places of the current
     component and the immediate transitions identified in the previous
   step
(b) No:
   i. Retrieve the engineer identifiers from the corrective class instance
   ii. Create a place to represent that there is an engineer available for the
       component. This place follows the following format:
       \( \text{component.componentType.componentID.maint.a} \)
   iii. Create a place to represent that maintenance has completed on the
       component. This place follows the following format:
       \( \text{component.componentType.componentID.maint.c} \)
   iv. Set the token to value to 1 in the engineer available place created in 4(b)ii.
   v. Retrieve the immediate transitions linked between failed places and the
      repairable place of the component
   vi. Create a single-headed arc between the engineer available place created in
      4(b)ii and each immediate transition
   vii. Retrieve the time to repair transitions
   viii. Create a single-headed arc between the time to repair transitions and the
      maintenance complete place created in 4(b)iii
   ix. For every engineer assigned to this component complete the following:
      A. Create a place to represent that the engineer is carrying out main-
         tenance. This place identifier has the following format: \( \text{component.maint.corr.engID.up} \)
      B. Create a place to represent that the engineer is not currently carrying
         out maintenance. The place identifier has the following format:
         \( \text{component.maint.corr.engID.down} \)
      C. Set the place token value to 1 for the place created in previous step
D. Create an immediate transition that will transition between the engineer carrying out maintenance place and the engineer not carrying out maintenance. The transition identifier follows the following format: \textit{component.transition.corr.engID.upToDown}

E. Create an immediate transition that will transition between the engineer not carrying out maintenance place and the engineer carrying out maintenance place. The transition identifier follows the following format: \textit{component.transition.corr.engID.DownToUp}

F. Create a single-headed arc between the maintenance complete place and the transition created in 4(b)ixD.

G. Create a single-headed arc between the engineer completing maintenance place and the transition created in 4(b)ixD.

H. Create a single-headed arc between the transition created in 4(b)ixD and the engineer not completing maintenance place.

I. Retrieve the failed places of the component.

J. For each of the failed places complete the following:
   \begin{itemize}
   \item Create an immediate transition will connect the place representing the engineer not completing maintenance and the engineer completing maintenance. The transition identifier follows the following format: \textit{component.transition.corr.engID.componentID_FailureValue}
   \item Create an inhibit arc between any places already created for other maintenance engineers as created in 4(b)ixB.
   \item Create a double-headed arc between the failed place and the transition created in 4(b)ixE.
   \item Create a single-headed arc between the transition created in 4(b)ixE and the maintenance available place created in 4(b)ii.
   \item Create a single-headed arc between the engineer not carrying out maintenance place created in 4(b)ixB and the transition created in 4(b)ixE.
   \item Create a single-headed arc between the transition created in 4(b)ixE and the engineer completing maintenance place created in 4(b)ixA.
   \end{itemize}

K. For each corrective plan left that has a common maintenance engineer complete the following:
   \begin{itemize}
   \item Retrieve the immediate transitions linked between failed places and the repairable place of the component.
   \end{itemize}
• For each failed place of the original component create an inhibit arc between the failed place and each transition identified in the previous step.

If there is more than one component that uses the same engineer then only a single place representing this engineer would be created. The appropriate links would then be created for each of the components maintained by the engineer.

Using the construction procedure above, three examples have been generated. The first, Figure 7.11, shows a single operational mode component under a corrective maintenance. The second, Figure 7.12, shows the construction of two components; one with a single operational mode and another with multiple operational modes. This is an example of two components sharing a single maintenance engineer. The third example, Figure 7.13, shows three repairable components within a system that are maintained by a system wide corrective maintenance cycle with two maintenance engineers.

7.4 Standby Systems

Standby systems are used as a method of redundancy within a system structure. There are three types of standby systems: cold, warm and hot. Each of these will be discussed in detail with examples of their Petri net equivalent. The procedures for the construction of each type will also be discussed.

7.4.1 File Input

Any standby components that exist within a system must be described as part of the system description to input into the software. This is dealt with within the simulation file (.sim) under the header STANDBY. An example of this can be seen in Figure 7.14. The first two declarations within the STANDBY header show that compB is in standby for compA and vice versa where compB is the first component in standby.
Figure 7.12: Petri net for multiple components maintained by a single maintenance engineer
Figure 7.13: Petri net for a system-wide corrective maintenance plan with two maintenance engineers

Figure 7.14: Declarations used within the STANDBY header within the simulation file
7.4. Standby Systems

Figure 7.15: The `standby` class

| class standby |
| -string componentId; |
| -vector<string> standbyIds; |
| -bool primary; |
| -string type; |

Figure 7.14 shows the header `STANDBY`, which is used to signify the next set of information to the software. The curly braces signify the beginning and the end of the standby information. Each line uses the following syntax:

```
compID *type(compID_1, compID_2, ..., compID_N);
```

Where a * is required if components can be in standby for one another. This identifies which component is in standby at the beginning of a simulation.

The syntax states that should component `compID` fail, component `compID_1` takes over; should `compID_1` fail then `compID_2` takes over and so on until no other components are available. If the system is designed such that, upon failure, the original component `compID` is repaired and placed into standby for the other components then another line would be required to show this in this section of the system description, e.g. `compID_1 type(compID_1, compID_2, ..., compID_N)`.

The Petri net is constructed according to the standby type (keywords: COLD, WARM and HOT). Each of the following sections shows an example of the construction procedure needed to generate the correct Petri net directly from the information above. At the point the standby section of the component/system is generated, the overall component Petri net has already been generated including the working to failed Petri net. The integration performed here simply connects the component to the standby components or systems.

### 7.4.2 System Storage

Each standby declaration as those seen in Figure 7.14 are held within a `standby` class instance. When each class instance is created it is held within the `maintenance` class instance. The `standby` class can be seen in Figure 7.15. The variable `type` would be COLD, WARM or HOT. The procedure for taking the information from the file and storing it within each `standby` class instance can be see in Figure 7.16.
The software has identified the header STANDBY within the simulation file.

Is the next string in the file "{"?
If yes, go to step 4.

Is the next string in the file "}
If yes, go to step 2.

Create a new standby class instance.

Get the next string, the component ID. Does the component exist in the system?
If yes, go to step 6.

Issue error to the user: Missing '}' from the declaration.

Add the component identifier to the standby class instance created in step 4.

Is the next string "*"?
If yes, go to step 7.

Issue error to the user: The component does not exist.

Is the next string ";"
If yes, go to step 8.

Issue error to the user: Missing '}' from the declaration.

Is the next string ":"
If yes, go to step 9.

Issue error to the user: Missing ":" from the declaration.

This is the primary component. Set primary to true.

Add the component to the standby class instance.

Is the next string "?"
If yes, go to step 10.

Issue error to the user: Missing ":" from the declaration.

Is the next string ";"
If yes, go to step 11.

Issue error to the user: Missing ":" from the declaration.

Figure 7.16: Flow chart for the storing of the standby components
7.4.3 Construction Procedure

The construction procedure for a standby system is given below:

1. Retrieve the component identifiers of those components in standby.
2. Retrieve the primary component’s CPN.
3. Retrieve the standby components’ CPNs.
4. For each standby component complete the following:
   (a) Create the place identifier (not the place) to represent the component within a standby state. The place identifier follows the following format: \( \text{component.componentType.componentID.state.standby} \)
   (b) Retrieve the time to failure transitions of the standby components.
   (c) If the standby component is of type COLD: Create an inhibit arc between the standby state place in 4a and each time to failure transition.
   (d) If the standby component is of type WARM: Create an inhibit arc between the standby state place in 4a and every other time to failure transition that is not the standby time to failure transition.
   (e) Create a new place to represent that the standby component is no longer in standby. The place identifier has the following format: \( \text{component.componentType.componentID.state.up} \).
   (f) If the standby component is of type WARM: Create an inhibit arc between the place created in the previous step and every other time to failure transition that is not the standby time to failure transition.
   (g) Does this component already have a standby place (a place with an identifier of the format seen in 4a)?
      i. Yes: Move on to the next step
      ii. No:
         A. Create the standby state place for the standby component. If this is not the primary component, set the token value for the place to 1.
         B. Create a standby ‘up’ place. The format for the identifier is as follows: \( \text{component.componentType.componentID.state.up} \). If this is not the primary component then set the place to have a token value of 1.
         C. Create a standby ‘down’ place. The format for the identifier is as follows: \( \text{component.componentType.componentID.state.down} \).
(h) Retrieve the working state place identifier and the failed state place identifiers of the standby component

(i) For each failed place identifier complete the following tasks:

i. Create a new immediate transition within the following transition identifier: component.componentType.componentID.standbyTransition.original_componentID.original_failValue

ii. Create a single-headed arc between the standby state place and the transition created in 4(i)i.

iii. Create a double-headed arc between the failed place identifier and the transition created in 4(i)i.

iv. Create a single-headed arc between the transition created in 4(i)i and the up place created in 4(g)iiB.

(j) Retrieve the working state place identifier and the failed state place identifiers of the original component

(k) For each failed place identifier complete the following tasks:

i. Create a new immediate transition within the following transition identifier:

   component.componentType.componentID.standbyTransition.downToStandby

ii. Create a single-headed arc between the down place created in 4(g)iiC and the transition created in 4(k)i.

iii. Create a single-headed arc between the transition created in 4(k)i and the standby state place.

iv. Create a double-headed arc between the working state place of the component and the transition created in 4(k)i.

(l) Retrieve the DT transitions for the standby component.

(m) For each transition create a inhibit arc between the standby state place and each transition.

7.4.4 Cold Standby

During the construction process cold standby components simply add inhibit arcs to stop the time to failure transitions from aging. When a cold standby component becomes operational then the time to failure transition begins to age. An example of a cold standby connection can be seen in Figure 7.17. Figure 7.17 shows two power supplies in cold standby. PS1 is the primary component and PS2 is the initial standby component.
7.4. Standby Systems

![Diagram of standby systems](image)

**Figure 7.17: Example of power supplies in cold standby**

7.4.5 Warm Standby

When work to fail Petri nets are created another time to failure transition is created that this is the rate of failure of the component whilst in warm standby. The relationship between working and standby failure rate can be seen in Figure 7.18. When entering this within the simulation file the following format is used:

```
componentID failure_value(or mode) standby failure_distribution(parameter(s));
```

An inhibit arc is used on the operational time to failure transition to ensure this does not age. Once operational the operational time to failure transition becomes enabled. An example of a warm standby connection is seen in Figure 7.19.

7.4.6 Hot Standby

As hot standby components fail at the same rate as if they were in operation no further transitions or inhibit arcs are required on the time to failure transitions. An example of a hot standby component can be seen in Figure 7.20.
Figure 7.18: Example of work to fail to repair relationship for a single mode component in warm standby

Figure 7.19: Example of power supplies in warm standby
Voting systems can also be used as a method of redundancy and therefore have been included in the software. This section describes how the user defines a voting system and how the software takes this information, stores it and generates the appropriate model from it.

### 7.5.1 File Input

The user defines a voting system within the simulation file (.sim). The header `VOTING` is used to identify the following declarations. To declare a voting system the following syntax is used:

\[
\text{Number\_Working}(\text{CompID}_1, \text{CompID}_2, \cdots, \text{CompID}_N);
\]

The \(N\)th component has a limit of five components. The reason for this is the model becomes very large and would slow the software. It has been defined in the software to limit this value for performance considerations only. An example of this written within the simulation file has been given in Figure 7.21.
7.5.2 System Storage

Each voting system defined in the simulation file is created in an instance of the `voting` class. The `voting` class can be seen in Figure 7.22. The procedure for storing information on the voting plans is shown in the flow chart in Figure 7.23.

7.5.3 Construction Procedure

The construction procedure for generating the Petri net for voting systems is listed below:

1. Retrieve the list of components associated with the voting plan
2. Retrieve the CPN for each of the components associated to the voting plan
3. Retrieve the working state place for each component
4. For each working state value (this is applicable to failed state values) for all components create a new place with the identifier of the format:
   
   \[ \text{component.compID}_1\_\text{compID}_2\_\cdots\_\text{compID}_N.\text{componentType.portName.value.} \]

5. Create a list of combinations of components working/failed.
6. For each place created in 4 complete the following:
   
   (a) For every combination found in 5 create an immediate transition with the following identifier:
   
   \[ \text{component.VOTING.compID}_1\_\text{compID}_2\_\cdots\_\text{compID}_N.\text{portValue.} \]

   \[ \text{combinationListNumber}. \]
7.5. Voting Systems

Figure 7.23: Flow chart for the storing of the voting information
State = W

State = W

State = W

State = F

State = F

State = F

A

B

C

A_B_C

DT transitions

IN1 = type 1

IN1 = type 2

OUT1 = type 1

OUT1 = type 2

Figure 7.24: An example of a 2-out-of-3 voting system Petri net

(b) For each voting component entered in the file, in the order listed, consider every combination created in 5. Create an inhibit arc between the component’s working place and any combination, represented by a transition, that includes only components further down in the voting list. This is only applicable to working places.

(c) Create a single-headed inhibit arc between the immediate transition created in 6a and the place created in 4.

(d) For each combination, represented by a transition, create a double-headed arc between the component places identified in 3, where the component is part of the combination.

7. Repeat from step 3 for failed identifiers.

8. During a later part of the model creation only one decision table is used between the voting components. The arcs between the one of the component’s working/failed states are transferred to the voting component’s working/failed states created in 4.

Due to the nature in which the Petri net has been created there is a need to control certain aspects of the Petri net to ensure multiple firing of tokens does not occur across the Petri net. Step 6b is used to limit the firing of the voting Petri net by creating inhibit arcs to the later transitions. An example of a two-out-of-three voting system can be seen in Figure 7.24.
7.6 Mission Abort

Within phased-mission systems there is scope to include the aborting of a mission based on a component failing within a given phase. An example of such an occurrence is an aircraft with 4 engines and one fails during flight, if this were to happen, then the likelihood is that the plane would be diverted to the nearest airport, particularly if there is a fear of further engine failure. The software caters for this eventuality.

7.6.1 File Input

There is a section defined in the simulation file (.sim) that covers mission abort due to a specific component failure within a given phase. This can be found under the \textit{ABORT} header. The user can stipulate the information as follows:

\begin{verbatim}
componentID(component_failure_mode) : (Phase_number);
\end{verbatim}

When mission abort is considered there is additional information within the final results file output from the simulation showing the number of aborted missions (as well as the number of failed missions). An example of the representation can be seen in Figure 7.25.

The example given states that should component \textit{S1} be in the state \textit{F_open} at any point in phase 1 then the mission is aborted. Similarly if the component \textit{A} is in the state \textit{F} at any point during phase 3 or phase 4 the mission is aborted.

Figure 7.25: File format for the ABORT header within the simulation file

7.6.2 System Storage

The information obtained from the simulation file is stored within a separate class instance called \textit{abortMission}. The \textit{abortMission} class can be seen in Figure 7.26. The process for parsing this section of the simulation file can be seen in Figures 7.27 and 7.28.

7.6.3 Construction Procedure

To construct the Petri net that represents the aborting of a mission is completed during the procedure to create the PPN. The construction of the Petri net to account for an
aborted mission is described below:

1. Create a place representing mission abort. This would be given the place identifier \textit{phase.abort}.

2. Locate all phase transitions that are not related to mission abort.

3. For each transition identified in 2 create an inhibit arc from the place created in 1 to each transition.

4. For each abort condition found from the simulation file complete the following:

   (a) Retrieve the phases that are associated with the abort condition

   (b) Retrieve the phase places that are associated to the phase numbers retrieved in 4a.

   (c) Create an immediate transition with the identifier \textit{component.abort.componentID.component\_fail\_value.phase\_number}

   (d) Create an arc for the following:

      i. A single-headed arc between the phase place identified in 4b and the new transition created in 4c.

      ii. A single-headed inhibit arc between the transition created in 4c and the abort place created in 1.

   (e) Retrieve the place that represents the component fail place value

   (f) Create an arc between the place located in 4e and the transition created in 4c.

Using the example in Figure 7.26, an example of the presentation of the abort conditions can be seen in Figure 7.29. In step 4(d)ii inhibit arcs are created between the abort phase place and the other transitions. The reason for this is to halt any other movement within the PPN once the mission has entered an abort phase.
Figure 7.27: The first part of the software algorithm to populate the system with abort conditions
Figure 7.28: The second part of the software algorithm to populate the system with abort conditions

Figure 7.29: Example representation of the abort process within a PPN
7.7 Simulating a Repairable System

The previous sections have dealt with the repairable aspects of the system in terms of how these are given in files to the software and how the software interprets the files in order to construct each model. This section describes how these repairable elements affect the way the software simulates a given repairable system and mission. The general simulation process as discussed in section 6.3.3.1 still applies to systems that are repairable. There are a number of new additional steps used in order to process some of the repairable aspects of a system. This is particularly important for preventative maintenance plans as these require a simulation of just the maintenance rather than the whole system. This section discusses the additional steps needed in order to simulate a repairable system. During the writing of the code to incorporate the repairable system aspects, much of the simulation code was altered and as a result became more efficient. These changes were already incorporated into the algorithms and procedures seen in Chapter 6.

7.7.1 Simulation Algorithm

The simulation algorithm for a repairable system builds on the algorithm described in section 6.3.3.1. In step 1 within the algorithm in section 6.3.3.1 there are a number of places and transitions that are located and their positions stored. For the repairable case the following places and transition positions must also be stored:

- Locate standby place positions
- Locate repair place positions
- Locate the positions of the time to repair transitions and the time between maintenance cycle transitions.
- If there are any preventative maintenance plans then the location of the simulation.maintenance place created as part of the preventative maintenance Petri net should also be stored.

All these place and transition locations are necessary in order to make the software more efficient. This is particularly useful when identifying the simulation end and the change in the system state.

Step 8 in the algorithm described in section 6.3.3.1 describes what occurs when a simulation of the mission has occurred. For a repairable mission this changes to the following:

1. Find the earliest time to failure and run a single simulation of the mission
2. If the mission was successful and the simulation end time has not been reached complete the following:

   (a) Test if there is a preventative maintenance cycle due at the end of the mission.

   (b) Yes: The software simulates the maintenance process. Go to step 1.

   (c) No: Re-run the mission to test if the component would cause a failure in the next mission. If this is again successful return to 1, otherwise go to step 3.

3. If the mission was not successful complete one of the following:

   • If there are no further simulations to run, end the simulation
   • If there are further simulations change the timed transitions and restart the process from step 1.

4. After maintenance has occurred simulate the model again. Return to step 1.

### 7.7.2 Simulation of the model

Every item completed as part of the simulation of the model seen in section 6.3.3.2 is still applicable for the repairable case. There a few additions to this part of the process which have been included below:

- To test for simulation end when considering the *earliest time* the down time created as part of the preventative maintenance cycle is accounted for.

- During the process shown in Figure 6.19 there is a further step after (3g) where the timed transitions, namely the time to failure and time to repair transitions are re-initialised.

### 7.7.3 Simulating Transitions

There is a single difference within this process and that is in the creation of the $\Sigma$ matrix. This is split depending on whether a maintenance cycle is in progress or the normal operational mission. For the creation of the $\Sigma$ matrix during the maintenance cycle the only transitions that are considered during the process is the time to repair transitions and the preventative time between cycle transitions.

### 7.8 Repairable Bulb System

#### 7.8.1 Introduction

To demonstrate the capabilities of the software for the processing of a repairable system, a bulb system was used. The bulb system consists of four major components: a bulb, a
toggle switch and two power supplies, one of which is in standby for the other. An operator is also used as part of the system, but is assumed not to be capable of failure. Figure 7.30 demonstrates the system considered.

7.8.1.1 System Process

The system is initiated by switching the toggle switch from an open operational mode to a closed operational mode. The bulb, $B$, turns ON. After a time of 20 hours the operator opens the toggle switch, $S$, and the bulb, $B$, turns OFF.

7.8.1.2 Initial Conditions of the System

All the components within the bulb system are assumed to be working from time $=0$. The component toggle switch, $S$, has a starting mode of open.

7.8.2 System Description

7.8.2.1 System Topology

The bulb system schematic in Figure 7.30 was transformed into the topology diagram seen in Figure 7.31. The power supply, $PS2$, is in standby for the power supply, $PS1$. Even when components are in standby the connections to these components must still be stated in the system topology.

Circuit Lists

There are two circuits that are present within this system:
Figure 7.31: System topology diagram for the bulb system

1. \{PS1, J2, S, B, J1, PS1\}

2. \{PS2, J2, S, B, J1, PS2\}

7.8.2.2 Component Decision and Operational Mode Tables

Each of the components were described using DTs and where applicable, OMTs. The DTs and OMTs for the bulb system can be found in Tables 7.1-7.5. The under repair state, \( UR \), has been included within the DTs and OMTs. This should be included to show how the component behaves when the component is under repair. This is particularly important for components under a corrective maintenance plan.

Table 7.1: Decision table for the component Bulb

<table>
<thead>
<tr>
<th>In 1</th>
<th>state</th>
<th>Out 1</th>
<th>Out 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>W</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>NC</td>
<td>–</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>F</td>
<td>NC</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>UR</td>
<td>NC</td>
</tr>
</tbody>
</table>
Table 7.2: Decision table for the Operator

<table>
<thead>
<tr>
<th>Time</th>
<th>In 1</th>
<th>state</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>OFF</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>0 &lt; t ≤ 20</td>
<td>OFF</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>0 &lt; t &lt; 20</td>
<td>ON</td>
<td>W</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>ON</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 7.3: Decision table for the component power supply

<table>
<thead>
<tr>
<th>In 1</th>
<th>State</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>UR</td>
</tr>
</tbody>
</table>

Table 7.4: Decision table for the component Toggle Switch

<table>
<thead>
<tr>
<th>in 1</th>
<th>Mode</th>
<th>Out 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>Closed</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>Open</td>
</tr>
<tr>
<td>3</td>
<td>NC</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 7.5: Operational Mode Transition Table for the component toggle Switch

<table>
<thead>
<tr>
<th>Mode 1 Command (In1)</th>
<th>State</th>
<th>Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Closed</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>Closed</td>
<td>CL</td>
</tr>
<tr>
<td>3</td>
<td>Closed</td>
<td>OP</td>
</tr>
<tr>
<td>4</td>
<td>Closed</td>
<td>NA</td>
</tr>
<tr>
<td>5</td>
<td>Closed</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>Open</td>
<td>OP</td>
</tr>
<tr>
<td>8</td>
<td>Open</td>
<td>CL</td>
</tr>
<tr>
<td>9</td>
<td>Open</td>
<td>NA</td>
</tr>
<tr>
<td>10</td>
<td>Open</td>
<td>–</td>
</tr>
</tbody>
</table>

7.8.2.3 Component Failure and repair data

Each of the components apart from the operator are repairable components. The failure and repair data for the bulb system can be seen in Table 7.6. Each of the components fails by an exponential distribution.
7.8.3 Mission Description

The system process described in section 7.8.2 has been transformed into the phase transition table seen in Table 7.7. The phases of the mission have been described below:

- Phase 1: System start-up
- Phase 2: Bulb is lit
- Phase 3: Mission Success

The failure phases are as follows:

- Phase 4: System fails to start
- Phase 5: System fails and the Bulb turns OFF
- Phase 6: System fails and bulb remains lit

7.8.4 Maintenance Plan

Each component within the system is maintained through a corrective maintenance cycle and there is an engineer assigned to each component. The component PS2 is in standby for the component PS1. If PS1 fails and then repaired this component then becomes the standby component for PS2.

7.8.5 Petri Net Models

7.8.5.1 Component Petri Nets and System Petri Net

The CPN created from the DTs and OMT seen in Tables 7.1-7.4 can be seen in Figures 7.32a-7.32d. There is no change in the way that these component tables are used to generate the CPN models.

After generating the CPNs the SPN was built by connecting the components by using the system topology. It should be noted that there is no special connection between standby components. The SPN for this system can be seen in Figure 7.33.

Table 7.6: Failure and repair data for the components of the bulb system

<table>
<thead>
<tr>
<th>Component Identifier</th>
<th>Failure Rate</th>
<th>Repair Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0.001</td>
<td>0.1</td>
</tr>
<tr>
<td>B</td>
<td>0.005</td>
<td>0.1</td>
</tr>
<tr>
<td>PS1</td>
<td>0.0067</td>
<td>0.02</td>
</tr>
<tr>
<td>PS2</td>
<td>0.0067</td>
<td>0.02</td>
</tr>
</tbody>
</table>
7.8. Repairable Bulb System

Table 7.7: Phase transition table for the bulb system

<table>
<thead>
<tr>
<th>Time</th>
<th>From Phase</th>
<th>To Phase</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>δ</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>δ</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

(a) CPN for the component Bulb, B

(b) CPN for the component Operator, OP

(c) CPN for the component Toggle Switch, S

(d) CPN for the component Power Supply, PS1 and PS2

Figure 7.32: CPNs for the components of the bulb system
Figure 7.33: SPN of the bulb system
7.8. Repairable Bulb System

7.8.5.2 Phase Petri Net

By using the phase transition table in Table 7.7 the PPN was generated as seen in Figure 7.34.

7.8.6 Validation

The bulb system discussed above was used to verify that the software could cater for repairable systems. In order to verify the system an analytical solution was required. The modelling method used was Markov modelling as discussed in section 2.3.1. The Markov approach was the simplest method to provide an analytical solution. A number of assumptions were made and are discussed in the following section.

7.8.6.1 Assumptions

1. Switch $S$ cannot fail in operational mode open, $FOP$. The switch changes from open to close in a discrete phase and only returns to this mode at the end of the mission and therefore does not spend any time in this mode for it to age. As a result the markov model only considers the failure mode $FCL$.

2. The component operator is assumed to be perfect, i.e. it cannot fail.

7.8.6.2 Markov Model

The Markov model states can be found in Table 7.8. The full Markov diagram can be seen in Figure 7.35. By using the method described by Clarotti et al (1980) and the Markov diagram shown the analytical values were calculated. The transition matrix for the Markov model can be seen in equation 7.8.1. Using this and the initial conditions as

---

Figure 7.34: PPN of the bulb system
seen in equation 7.8.2 the analytical values for the bulb system were calculated for each phase of the mission.

Table 7.8: Markov model states for repairable bulb system

<table>
<thead>
<tr>
<th>State</th>
<th>B</th>
<th>S</th>
<th>PS1</th>
<th>PS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>W</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>W</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>W</td>
<td>FCL</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>FCL</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>7</td>
<td>W</td>
<td>W</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>8</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>FCL</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>FCL</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>W</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>W</td>
<td>W</td>
<td>F</td>
</tr>
<tr>
<td>13</td>
<td>W</td>
<td>FCL</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>14</td>
<td>W</td>
<td>FCL</td>
<td>W</td>
<td>F</td>
</tr>
<tr>
<td>15</td>
<td>W</td>
<td>W</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>F</td>
<td>FCL</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>17</td>
<td>F</td>
<td>FCL</td>
<td>W</td>
<td>F</td>
</tr>
<tr>
<td>18</td>
<td>F</td>
<td>W</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>19</td>
<td>W</td>
<td>FCL</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>20</td>
<td>F</td>
<td>FCL</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Figure 7.35: Markov Model of the Bulb System
\begin{alignat}{10}
&\lambda_1 P_{11} + \lambda_2 P_{12} + \lambda_3 P_{13} + \lambda_4 P_{14} + \lambda_5 P_{15} + \lambda_6 P_{16} + \lambda_7 P_{17} + \lambda_8 P_{18} &= 0 \\
&-\lambda_1 P_{21} - \lambda_2 P_{22} - \lambda_3 P_{23} - \lambda_4 P_{24} - \lambda_5 P_{25} - \lambda_6 P_{26} - \lambda_7 P_{27} - \lambda_8 P_{28} &= 0 \\
&\lambda_1 P_{31} + \lambda_2 P_{32} + \lambda_3 P_{33} + \lambda_4 P_{34} + \lambda_5 P_{35} + \lambda_6 P_{36} + \lambda_7 P_{37} + \lambda_8 P_{38} &= 0 \\
&-\lambda_1 P_{41} - \lambda_2 P_{42} - \lambda_3 P_{43} - \lambda_4 P_{44} - \lambda_5 P_{45} - \lambda_6 P_{46} - \lambda_7 P_{47} - \lambda_8 P_{48} &= 0 \\
&\lambda_1 P_{51} + \lambda_2 P_{52} + \lambda_3 P_{53} + \lambda_4 P_{54} + \lambda_5 P_{55} + \lambda_6 P_{56} + \lambda_7 P_{57} + \lambda_8 P_{58} &= 0 \\
&-\lambda_1 P_{61} - \lambda_2 P_{62} - \lambda_3 P_{63} - \lambda_4 P_{64} - \lambda_5 P_{65} - \lambda_6 P_{66} - \lambda_7 P_{67} - \lambda_8 P_{68} &= 0 \\
&\lambda_1 P_{71} + \lambda_2 P_{72} + \lambda_3 P_{73} + \lambda_4 P_{74} + \lambda_5 P_{75} + \lambda_6 P_{76} + \lambda_7 P_{77} + \lambda_8 P_{78} &= 0 \\
&-\lambda_1 P_{81} - \lambda_2 P_{82} - \lambda_3 P_{83} - \lambda_4 P_{84} - \lambda_5 P_{85} - \lambda_6 P_{86} - \lambda_7 P_{87} - \lambda_8 P_{88} &= 0 \\
&\lambda_1 P_{91} + \lambda_2 P_{92} + \lambda_3 P_{93} + \lambda_4 P_{94} + \lambda_5 P_{95} + \lambda_6 P_{96} + \lambda_7 P_{97} + \lambda_8 P_{98} &= 0 \\
&-\lambda_1 P_{101} - \lambda_2 P_{102} - \lambda_3 P_{103} - \lambda_4 P_{104} - \lambda_5 P_{105} - \lambda_6 P_{106} - \lambda_7 P_{107} - \lambda_8 P_{108} &= 0 \\
&\lambda_1 P_{111} + \lambda_2 P_{112} + \lambda_3 P_{113} + \lambda_4 P_{114} + \lambda_5 P_{115} + \lambda_6 P_{116} + \lambda_7 P_{117} + \lambda_8 P_{118} &= 0 \\
&-\lambda_1 P_{121} - \lambda_2 P_{122} - \lambda_3 P_{123} - \lambda_4 P_{124} - \lambda_5 P_{125} - \lambda_6 P_{126} - \lambda_7 P_{127} - \lambda_8 P_{128} &= 0 \\
&\lambda_1 P_{131} + \lambda_2 P_{132} + \lambda_3 P_{133} + \lambda_4 P_{134} + \lambda_5 P_{135} + \lambda_6 P_{136} + \lambda_7 P_{137} + \lambda_8 P_{138} &= 0 \\
&-\lambda_1 P_{141} - \lambda_2 P_{142} - \lambda_3 P_{143} - \lambda_4 P_{144} - \lambda_5 P_{145} - \lambda_6 P_{146} - \lambda_7 P_{147} - \lambda_8 P_{148} &= 0 \\
&\lambda_1 P_{151} + \lambda_2 P_{152} + \lambda_3 P_{153} + \lambda_4 P_{154} + \lambda_5 P_{155} + \lambda_6 P_{156} + \lambda_7 P_{157} + \lambda_8 P_{158} &= 0 \\
&-\lambda_1 P_{161} - \lambda_2 P_{162} - \lambda_3 P_{163} - \lambda_4 P_{164} - \lambda_5 P_{165} - \lambda_6 P_{166} - \lambda_7 P_{167} - \lambda_8 P_{168} &= 0 \\
&\lambda_1 P_{171} + \lambda_2 P_{172} + \lambda_3 P_{173} + \lambda_4 P_{174} + \lambda_5 P_{175} + \lambda_6 P_{176} + \lambda_7 P_{177} + \lambda_8 P_{178} &= 0 \\
&-\lambda_1 P_{181} - \lambda_2 P_{182} - \lambda_3 P_{183} - \lambda_4 P_{184} - \lambda_5 P_{185} - \lambda_6 P_{186} - \lambda_7 P_{187} - \lambda_8 P_{188} &= 0 \\
&\lambda_1 P_{191} + \lambda_2 P_{192} + \lambda_3 P_{193} + \lambda_4 P_{194} + \lambda_5 P_{195} + \lambda_6 P_{196} + \lambda_7 P_{197} + \lambda_8 P_{198} &= 0 \\
&-\lambda_1 P_{201} - \lambda_2 P_{202} - \lambda_3 P_{203} - \lambda_4 P_{204} - \lambda_5 P_{205} - \lambda_6 P_{206} - \lambda_7 P_{207} - \lambda_8 P_{208} &= 0
\end{alignat}
7.8. Repairable Bulb System

\[ P(0) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \] (7.8.2)

7.8.6.3 Single Mission Condition

The software was used to simulate the condition of a single mission in order to validate the model generated using the algorithms discussed in sections 6.3.3 and 7.7. As a consequence of running the simulation for a single mission the system preventative maintenance cannot be tested for this simulation set. Each component therefore was set to be part of a corrective maintenance policy. The simulation was run for 5,000 simulations and the results can be seen in Table 7.9. To prove repeatability the simulation was re-run for a further 5,000 simulations. The results of which can be seen in Table 7.10. Each set of results remain within the ±5% tolerance.

Table 7.9: Bulb system simulation results for 5,000 simulations

<table>
<thead>
<tr>
<th>Phase Number</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytical</td>
<td>0.12</td>
</tr>
<tr>
<td>Simulation</td>
<td>0.1222</td>
</tr>
<tr>
<td>Difference(%)</td>
<td>1.83</td>
</tr>
</tbody>
</table>

Table 7.10: Second set of simulation results for the bulb system for 5,000 simulations

<table>
<thead>
<tr>
<th>Phase Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytical</td>
</tr>
<tr>
<td>Simulation</td>
</tr>
<tr>
<td>Difference(%)</td>
</tr>
</tbody>
</table>

Figure 7.36 shows the results obtained from both simulation runs together with the error margins and the analytical value calculated using the Markov model discussed in Section 2.3.1. The values remain within the ±5% margins after approximately 3,700 simulations.

7.8.6.4 Analysis

The simulation results showed good convergence upon the analytical values calculated using the Markov model in Figure 7.35 after approximately 3,700 missions during the
convergence study. The simulation results also showed that the values are repeatable to give the approximate same value.

7.9 Summary

This chapter has discussed the main repairable aspects that the software has been designed to handle. The corrective and preventative maintenance policies were the main aspects of a repairable system. The user input to define a repairable aspect, such as the maintenance policies, are easily implemented. There is not much more user information required for the software to cater for a repairable component/system as the software abstracts this process from them. The hardest aspect to implement within the software was the simulating of a preventative maintenance cycle.
Chapter 8

Conclusion and Further Work

Contents

8.1 Conclusion ................................................................. 243
8.2 Further Work ............................................................... 246
  8.2.1 Optimisation Study ................................................... 246
  8.2.2 Minimal Cut Sets ..................................................... 246
  8.2.3 Automatic Generation of the System Structure File .......... 247
  8.2.4 Multiple Interacting Systems .................................... 247

8.1 Conclusion

This thesis began by considering different modelling methods that were available that could potentially be used as part of the software. The modelling method was required to be versatile with the capability of being applied to both non-repairable and repairable systems. Modelling phased-mission systems adds a further level of complexity; specific methods applied to such systems were discussed to identify any methods that could be taken forward as part of the solution. After taking these criteria into account, Petri nets presented the most suitable modelling method. The main reason for this choice was due to the versatile nature of Petri nets. As was seen in section 1.4.3.2 Petri nets can be applied in a wide range of cases, including direct model conversions from other reliability modelling methods. Petri nets also stood out for the ability to model different aspects of a system and the mission undertaken, simultaneously modelling individual component states and more abstract aspects of the system such as progress through the current mission.

Literature considered in chapter 2 provided a grounding in current methods relating to phased-mission systems, including phase fault trees and repairable vs. non-repairable systems. The simulation process discussed in section 2.3.2 was of particular interest in relation to the simulation of the model, as discussed later in the thesis. The third chapter gave an overview of some of the methods used in automating the process of creating reliability models. Most automated methods were based around fault trees. As fault trees are unable to model dependencies required for repairable systems they were deemed
unsuitable as a modelling method for the software. Most of the methods describe taking an input file of the model and then using this to generate the data. As the software was required to take some form of file input from the user about the system, its components and the mission, this was of particular interest. One method that stood out was that of Henry and Andrews (1997) which takes in a system description from an AutoCAD diagram, decision tables and diagraph. After stating the top event for the system the fault tree was generated based on information in the input files. This method did not generate the data, but the files of the fault trees that could be used by commercial software to generate the data. The ability to model the component states using decision tables was of particular interest as this modelled the behaviour of individual components and provided a relatively intuitive format for component description. Further to decision tables, state transition tables provided a method of showing the behaviour of components with multiple operational modes. Combining these as part of the component description provided a solid foundation for the software.

The next chapters detailed the way in which Petri nets, decision tables and state transition tables were taken forward and used as the basis for the software. Chapter 4 detailed the method by which the system and mission description were broken down into simple input files. The methods of taking this information and transforming it into the four Petri net types (Component Petri nets, System Petri nets, Circuit Petri nets and Phase Petri nets) were discussed. The idea of breaking the Petri nets up into individual sections and components made the construction procedure for each type of Petri net self-contained and manageable. The biggest challenge in this work was to abstract as much of the model generation process as possible from the user. This ensures that the user is required to give only minimal information and requires the software to infer other information and completely automate the model generation process. This is a key aspect of the novel work presented in this thesis.

Once the process for model generation had been established it was applied to a pressure tank system in chapter 5. This system served to illustrate the key concepts involved in the model generation and was an important aspect of validating the process. This system was later used in simulations to validate the software for the non-repairable case.

Chapter 6 discussed in more detail the structure and algorithms involved in the implementation of this large and complex piece of software. The chapter can be divided into the three elements of the software: the input files written by the user, the model generation process and the simulation process. The chapter demonstrates each of the file types required from the user by showing a step-by-step guide of how each is formatted. Each file type requires minimal information for the software to build and simulate the model. The input files are simple to construct where the only complexity is understanding how a new component behaves. A function to create the circuit lists through recursive
8.1. Conclusion

exploration of the system topology was generated for this reason. This is a particularly useful function in that it only needs the information stored within the component table and the system topology files. These lists then form the basis of the Circuit Petri nets.

The simulation process discussed is specific to the model that has been generated and therefore includes unique steps that are not seen in other simulators. One particular step is the need to wipe clean the memory of the Petri net when there is a change of component and therefore system state. The simulator allows the user to specify the number of simulations as well as the duration of each simulation. This means that the simulator supports multiple consecutive missions performed by the same system, making it more flexible in its application. The simulator outputs detailed failure data for components, phases, missions and the overall system. The results of the simulations served to validate the software as a whole; from user input files through the unique model generation methods to the simulation that yields the final results. The results show an agreement with the analytical solution to within a tolerance of ±5%. This showed convergence on the analytical values calculated using phase fault trees within a few thousand simulations. The run time for these results was approximately 2.5-3 days to complete the 10,000 simulations on a typical laptop computer. It is likely that this could be reduced by an order of magnitude with code optimisations. The time taken to generate the Petri net model itself is in the order of tens of seconds, with execution time expected to rise in line with system size and complexity. What would usually take an individual days or weeks to complete takes less than a minute through this automated process.

The final chapter discusses how this software was further developed to cater for repairable components, adding a further layer of complexity. The chapter discusses the different maintenance plans that can be applied at either a component level or a system-wide event. By incorporating the preventative maintenance plan, the software simulation needed to change to account for the system being repaired outside the mission space. The software also caters for standby components which allows a user to model redundancy within a system. This would aid a design engineer in identifying to what extent providing a standby component for one prone to failure would increase the success rate of their system.

For further validation and to test the repairable case, the software was further applied to a bulb system. The simulation results obtained from the software showed that the software could produce results again within a tolerance of ±5%. Following minor simulation code optimisations, the simulation time for the 5000 simulations carried out for this system took less than 12 hours to produce. Multiple separate instances of the software can be run in parallel in order to run a number of design solutions at the same time. Once the files are generated, a single command performs the model generation and simulation process unaided.
This thesis has demonstrated a novel approach to generating a reliability model for phased-mission systems using Petri nets. The approach requires a simple expression of the system structure, its components and the mission the system is to undertake. This allows engineers with little or no experience in the field of risk and reliability to perform effective analyses of complex systems. From these inputs, a full model of the system is generated. A software implementation of the model generation process is discussed and used, in conjunction with a simulator written for the purpose. Further, the process has been extended to support systems with repairable and redundant components. The overall comparison of the results after 5,000 simulations with analytical results shows an error of less than ±5% in both repairable and non-repairable cases.

8.2 Further Work

8.2.1 Optimisation Study

The software has already undergone a small optimisation study in which the time for a simulation dropped to about a third of the previous value. It became apparent that using strings to identify places and transitions affected significantly the performance of the software, particularly when simulating the model, hence why a numeric identifier was introduced. The building process relies heavily on the string identifiers in order to connect the places and transitions together. If these were identified instead by numeric values then the time to build the model would also be reduced.

8.2.2 Minimal Cut Sets

Currently the software returns the overall system unreliability for the mission and each of the individual phases. Another output file lists the failures, repairs and maintenance that takes place during each simulation run of the mission. Another potential output from the software could be, for non-repairable systems, the minimal cut sets. This could be achieved manually by the user if so desired by setting components to fail in certain phases of the mission on each run to determine the minimal but sufficient component failures within the system that would cause the system to fail. There is no reason this process could not be automated using the software created. Instead of simulating the mission using generated time to failures the software could be programmed to move through each component in the topology and set the time to failure within a given phase and then run the mission. This could be repeated for each individual component and then combinations of components within a phase. If a component on its own could fail a given phase then this would not be tested further within this phase as it is proven that it is sufficient to cause a phase failure. This process would require a significant amount of computing time for larger systems when
considering combinations of the system’s components. This would also depend on whether
time is a factor in working these minimal cut sets out. If a user was willing to wait a
significant amount of time then this may still be acceptable.

This may even provide a faster method of generating the system reliability data for a
non-repairable systems of smaller systems. By using the minimal cut sets and applying the
work by La Band (2005). There is also no reason that the software could then produce an
output file that could be used within an application that can display the fault tree visually.
More study would be required into the feasibility of generating the analytical values from
the simulation model within a time that is satisfactory.

8.2.3 Automatic Generation of the System Structure File

From the files the user is required to generate as input to the software the system structure
file (.ss) has the greatest chance of error whilst generated by the user. The software will
detect most errors but the file would take time to create for larger systems. A method
to take a CAD diagram or Piping and Instrumentation Diagram (PID) and automatically
generate the file from these would save the user considerable time and effort, particularly
for larger systems. This is a method that has already been considered by Henry and
Andrews (1997) in which an AutoCAD diagram of the system is converted into a file to
show the component connections. This addition would reduce the overall time to generate
the simulation results, particularly for larger systems.

8.2.4 Multiple Interacting Systems

This software could easily be expanded to handle multiple systems that interact with each
other. This could be of particular interest if the user were designing an airport with a fleet
of aircraft. This would be done by allowing the user to enter multiple topology files into
the software and then generate a System Petri net for each system and then link them
together. If each system is undertaking a different mission this may be difficult to model
in a single Petri net model. This is due to the way in which the software simulates the time
of the mission. The Phase Petri net is used as a way of monitoring the time throughout
the simulation. If another Phase Petri net were included, any conflict between the two
would need to be resolved.
References


Schneeweiss, W. G. (1999), Petri nets for reliability modeling in the fields of engineering safety and dependability, LiLoLe-Verlag, Hagen.


A.1 Menu Interaction

The following shows how the user interacts with the software. The main menu of the software can be seen in Figure A.1. This has 5 options:

1. Loading the project file name into the software
2. Build the Petri net model
3. Simulate the built Petri net model
4. Output the system to a file and to screen
5. Delete the current system and mission

If the first option is selected then the user is presented with a prompt as seen in Figure A.2. This is where the user enters the project file (.prj), this must include the extension of the file.

The second option is selected by the user once the file has been loaded. There are no further prompts for the user.

The third option is selected by the user once the first two options have been selected. The user is required to enter two further details:

- The number of simulations
- The simulation length. This should be divisible by the mission length. E.g. if a mission is 20 hours long and the user wants to run the mission 5 consecutive times then the simulation time is 100.

The fourth option allows the current system stored in the software to be printed to file in a user friendly form. The file name is set as SystemFile.txt. This includes each individual component class instances including the decision and operational mode tables, the phase transition table and the circuit lists in the current system. The Component Petri net instances, System Petri net, Circuit Petri net and Phase Petri net descriptions are also included.
Figure A.1: Main Menu Screen

Figure A.2: Main Menu Option 1
A.1. Menu Interaction

Figure A.3: Main Menu Option 3
Appendix B

Pressure Tank System

B.1 Input Files

This section includes the files used to generate the Petri net based on the non-repairable Pressure Tank System

B.1.1 Project File

File Name: p2.prf
- - PROJECT pressure_tank_system

- - DT files:

contact.dt;
fuse.dt;
omotor.dt;
operator.dt;
powerSupply.dt;
pressureGauge.dt;
pump.dt;
relay.dt;
pushSwitch.dt;
toggleSwitch.dt;
tank.dt;
timerRelay.dt;
valve.dt;
junctionTwoIn.dt;
junctionOneIn.dt;
boundary.dt;

- - OMT files:

contact.omt;
pushSwitch.omt;
B.1.2 Component Files

This section covers the decision and operational mode table files of each component type within the pressure tank system.

B.1.2.1 Component Contact Decision and Operational Mode Table

File Name: contact.dt

DT contact
{
  in: in2, mode: mode3, out: out1;
  -, open, NC;
  NC, -, NC;
  C, closed, C;
}

File Name: contact.omt

OMT contact
{
  mode: mode1, in: in1, state: state1, mode: mode2;
  closed, -, F_closed, closed;
  closed, EN, -, closed;
  closed, DE, W, open;
  open, -, F_open, open;
  open, DE, -, open;
  open, EN, W, closed;
}
B.1.2.2 Component Fuse Decision Table

File Name: fuse.dt

DT fuse
{
in:in1, state:state1, out:out1;
C, W, C;
-, F, NC;
NC, -, NC;
}

B.1.2.3 Component Motor Decision Table

File Name: motor.dt

DT motor
{
in:in1, state:state1, out:out1, out:out2;
C, W, C, ON;
-, F, NC, OFF;
NC, -, NC, OFF;
}

B.1.2.4 Component Operator Decision Table

File Name: operator.dt

DT operator
{
time:time1, in:in1, state:state1, out:out1, out:out2, out:out3;
(t=0), LPR, W, CL, CL, CL;
(0<t<3), LPR, W, NA, NA, NA;
-, HPR, W, OP, NA, NA;
-, VHPR, W, NA, OP, NA;
-, -, F, NA, NA, NA;
}
B.1.2.5 Component Power Supply Decision Table

File Name: powerSupply.dt

DT power_supply
{
in:in1, state:state1, out:out1;
C, W, C;
-, F, NC;
NC, -, NC;
}

B.1.2.6 Component Pressure Gauge Decision Table

File Name: pressureGauge.dt

DT pressure_gauge
{
time:time1, in:in1, state:state1, out:out1;
(t<1), CONST, W, LPR;
(t<1), INC, W, LPR;
(1), CONST, W, LPR;
(1), INC, W, HPR;
-, DEC, W, LPR;
(1<t<=3), CONST, W, HPR;
(1<t<=3), INC, W, VHPR;
-, -, F_LOW, LPR;
-, -, F_HIGH, HPR;
-, -, F_VHIGH, VHPR;
}

B.1.2.7 Component Pump Decision Table

File Name: pump.dt

DT pump
{
in:in1, state:state1, out:out1;
ON, W, FL;
-, F, NFL;
B.1.2.8 Component Relay Decision Table

File Name: relay.dt

DT relay
{
    in:in1, state:state1, out:out1, out:out2;
    C, W, EN, C;
    -, F, DE, NC;
    NC, -, DE, NC;
}

B.1.2.9 Component Push Switch Decision and Operational Mode Table

File Name: pushSwitch.dt

DT push_switch
{
    in:in2, mode:mode3, out:out1;
    -, open, NC;
    NC, -, NC;
    C, closed, C;
}

File Name: pushSwitch.omt

OMT push_switch
{
    mode:mode1, in:in1, state:state1, mode:mode2;
    closed, -, F_closed, closed;
    closed, CL, -, closed;
    closed, NA, W, open;
    open, -, F_open, open;
    open, NA, -, open;
    open, CL, W, closed;
}
B.1.2.10 Component Toggle Switch Decision and Operational Mode Table

File Name: toggleSwitch.dt

\[
\text{DT toggle\_switch} \\
\{ \\
in:in2, mode:mode3, out:out1; \\
-, open, NC; \\
NC, -, NC; \\
C, closed, C; \\
\} \\
\]

File Name: toggleSwitch.omt

\[
\text{OMT toggle\_switch} \\
\{ \\
mode:mode1, in:in1, state:state1, mode:mode2; \\
closed, -, F\_closed, closed; \\
closed, CL, -, closed; \\
closed, OP, W, open; \\
closed, NA, -, closed; \\
open, -, F\_open, open; \\
open, OP, -, open; \\
open, NA, -, open; \\
open, CL, W, closed; \\
\} \\
\]

B.1.2.11 Component Tank Decision Table

File Name: tank.dt

\[
\text{DT tank} \\
\{ \\
time:time1, in:in1, in:in2, state:state1, out:out1, out:out2; \\
-, FL, open, W, CONST, FL; \\
-, FL, closed, W, INC, NFL; \\
-, NFL, closed, W, CONST, NFL; \\
(t=<1), NFL, open, W, CONST, NFL; \\
(1<t=<3), NFL, open, W, DEC, FL; \\
\} \\
\]
B.1. Input Files

(t=<1), - , - , F, CONST, NFL;
(1<t=<3), - , - , F, DEC, NFL;

B.1.2.12 Component Timer Relay Decision Table

File Name: timerRelay.dt

DT timer_relay
{
  time:time1, in:in1, state:state1, out:out1, out:out2;
  (t<1), C, W, EN, C;
  (t>=1), C, W, DE, NC;
  -, -, F, DE, NC;
  -, NC, -, DE, NC;
}

B.1.2.13 Component Valve Decision and Operational Mode Table

File Name: valve.dt

DT valve
{
  in:in2, mode:mode3, out:out1;
  - , closed, NFL;
  NFL, open, NFL;
  NFL, closed, NFL;
  FL, open, FL;
}

File Name: valve.omt

OMT valve
{
  mode:mode1, in:in1, state:state1, mode:mode2;
  closed, -, F_closed, closed;
  closed, CL, -, closed;
  closed, OP, W, open;
  closed, NA, -, closed;
open, -., F_open, open;
open, OP, -, open;
open, NA, -, open;
open, CL, W, closed;
}

B.1.2.14 Component Junction Decision Tables

File Name: junctionTwoIn.dt

DT junction_two_in
{
in:in1, in:in2, out:out1;
C, -, C;
-, C, C;
NC, NC, NC;
}

File Name: junctionOneIn.dt

DT junction_one_in
{
in:in1, out:out1, out:out2;
C, C, C;
NC, NC, NC;
}

B.1.2.15 Component Boundary Decision Table

File Name: boundary.dt

DT boundary
{
in:in1;
FL;
NFL;
}
### B.1.3 System Structure File

File Name: ss_pts.ss

```vhdl
SS SS_Structure1

begin
S1 : push_switch
port map(
in1 => OP_S1;
in2 => J4_S1;
out1 => S1_J1;
)

J1 : junction_two_in
port map(
in1 => CT_J1;
in2 => S1_J1;
out1 => J1_PS1;
)

PS1 : power_supply
port map(
in1 => J1_PS1;
out1 => PS1_S2;
)

S2 : toggle_switch
port map(
in1 => OP_S2;
in2 => PS1_S2;
out1 => S2_J2;
)

J2 : junction_one_in
port map(
in1 => S2_J2;
out1 => J2_TIM;
```
out2 => J2_R;
)

TIM : timer_relay
port map(
in1 => J2_TIM;
out1 => TIM_CT;
out2 => TIM_J3;
)

R : relay
port map(
in1 => J2_R;
out1 => R_CR;
out2 => R_J3;
)

J3 : junction_two_in
port map(
in1 => TIM_J3;
in2 => R_J3;
out1 => J3_J4;
)

J4 : junction_one_in
port map(
in1 => J3_J4;
out1 => J4_CT;
out2 => J4_S1;
)

CT : contact
port map(
in1 => TIM_CT;
in2 => J4_CT;
out1 => CT_J1;
)
CR : contact
port map(
in1 => R.CR;
in2 => FS.CR;
out1 => CR_M;
)

M : motor
port map(
in1 => CR_M;
out1 => M_PS2;
out2 => M_P;
)

PS2 : power_supply
port map(
in1 => M_PS2;
out1 => PS2_FS;
)

FS : fuse
port map(
in1 => PS2_FS;
out1 => FS_CR;
)

P : pump
port map(
in1 => M_P;
out1 => P_T;
)

T : tank
port map(
in1 => P_T;
in2 => V_T;
out1 => T_PG;
out2 => T_V;
V : valve
port map(
in1 => OP_V;
in2 => T_V;
out1 => V_B;
mode3 => V_T;
)

PG : pressure_gauge
port map(
in1 => T_PG;
out1 => PG_OP;
)

OP : operator
port map(
in1 => PG_OP;
out1 => OP_V;
out2 => OP_S2;
out3 => OP_S1;
)

B : boundary
port map(
in1 => V_B;
)

end

B.1.4 Phase Transition Table File

File Name: ptt_m4.ptt

ptt mission_1 {
time, from_phase, to_phase, condition;
0, 1, 2, CT.mode3=closed;
delta, 1, 8, CT.mode3=open;
1, 2, 3, T.out1=CONST;
-, 2, 8, T.out1=CONST;
-, 3, 4, V.mode3=open;
delta, 3, 5, V.mode3=closed;
3, 4, 9, T.out1=DEC;
-, 4, 8, T.out1=CONST;
delta, 5, 6, CR.mode3=open;
delta, 5, 7, CR.mode3=closed;
}

B.1.5 Simulation File

File Name: simulation_2.sim

--- This is the simulation file for the pressure tank system
--- This file contains:
--- 1.) The modes of components with an operational mode table.
--- 2.) The failure and repair distributions/rate.
--- 3.) The initiating component

SIM

--- Component Modes:
--- List as component_identifier mode = mode_type; e.g. S1 mode = OP;

COMPONENT MODES
{
S1 mode = open;
S2 mode = closed;
CT mode = open;
CR mode = open;
V mode = closed;
}

--- Failure distributions/rates
--- List as component_identifier distribution_type(parameter_1, parameter_2);
--- for WEIBULL distribution: component_identifier weibull(characteristic_Life, shape_parameter);
for EXPONENTIAL distribution: component_identifier exponential(mean);
- - for NORMAL distribution: component_identifier normal(standard_deviation, mean);
- - for time to failure: component_identifier ttf(rate);

FAILURE
{
S1 F_closed exponential(10);
S1 F_open exponential(10);
PS1 F exponential(1000.000);
S2 F_closed exponential(1000.000);
S2 F_open exponential(1.149689584);
TIM F exponential(1000.000);
R F exponential(10.000);
CT F_closed exponential(10.00);
CT F_open exponential(10.00);
CR F_closed exponential(4347.826087);
CR F_open exponential(4347.826087);
M F exponential(1000.000);
PS2 F exponential(1000.000);
FS F exponential(100.000);
P F exponential(10.000);
T F exponential(10000.000);
V F_closed exponential(33.333);
V F_open exponential(33.333);
PG F_LOW exponential(100.000);
PG F_HIGH exponential(100.000);
PG F_VHIGH exponential(100.000);
OP F exponential(10.000);
}

- - Initiating component
- - List as component_identifier port_name = value; e.g. OP out1 = CL;
INITIAL
{
OP out3 = CL;
}
B.2. Analytical Results

B.1.6 Setup File

File Name: setup.ini

- SETUP FILE -
INI
- - DEFAULT
- - Working state of components

working state :: W;

- - Wire link types current, no current used for the decision tables
- - List as Current, noCurrent

wire type :: C, NC;

B.2 Analytical Results

B.2.1 Single Mission

Table B.1 shows the phase reliability and unreliability values used within the analytical calculations seen in Chapter 6.

B.3 Simulation Results

B.3.1 Single Mission

Tables B.2 - B.4 show the simulation results for the single mission condition for the Pressure Tank System.

B.3.2 Multiple Missions

Table B.5 shows the simulation results for the testing of multiple consecutive missions undertaken by the Pressure Tank System.
Table B.1: Analytical values for the Unreliability and Reliability of each phase and each phase range

<table>
<thead>
<tr>
<th>ID</th>
<th>Failure Mode</th>
<th>Mean Time to Failure</th>
<th>Failure Rate</th>
<th>phase 2</th>
<th>phase 3</th>
<th>Unreliability, Q(t)</th>
<th>Reliability, R(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>phase 1-2</td>
<td>phase 1-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>phase 2</td>
<td>phase 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>phase 1-2</td>
<td>phase 1-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>phase 2</td>
<td>phase 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table continues with similar entries for different failure modes and parameters.
## Table B.2: Simulation Results for the single mission condition for the Pressure Tank System (Simulations 0-4000)

<table>
<thead>
<tr>
<th>Simulation Number</th>
<th>Number of failures</th>
<th>Phase Unreliability</th>
<th>Mission Unreliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>350</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>450</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>550</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>650</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>700</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>750</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>850</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>900</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>950</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1050</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1150</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1300</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1350</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1450</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1550</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1650</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1700</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1850</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1900</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1950</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2050</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2150</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2300</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2350</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2450</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2550</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2650</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2700</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2850</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2900</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2950</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3050</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3150</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3250</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3300</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3350</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3400</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3450</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3550</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3650</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3700</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3750</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3850</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3900</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3950</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### B.3. Simulation Results

The table above presents the results of the simulations for the single mission condition for the Pressure Tank System. Each row corresponds to a different simulation number, with columns indicating the number of failures, phase unreliability, and mission unreliability. The data are presented in a tabular format for easy review and analysis.
276

Appendix B. Pressure Tank System

Table B.3: Simulation Results for the single mission condition for the Pressure Tank System
(Simulations 4050-8000)

Simulation
Number

Phase 1

Number of failures
Phase 2
Phase 3

Phase 4

Phase 1

Phase Unreliability
Phase 2
Phase 3

Phase 4

Mission
Unreliability

4050
4100
4150
4200
4250
4300
4350
4400
4450
4500
4550
4600
4650
4700
4750
4800
4850
4900
4950
5000
5050
5100
5150
5200
5250
5300
5350
5400
5450
5500
5550
5600
5650
5700
5750
5800
5850
5900
5950
6000
6050
6100
6150
6200
6250
6300
6350
6400
6450
6500
6550
6600
6650
6700
6750
6800
6850
6900
6950
7000
7050
7100
7150
7200
7250
7300
7350
7400
7450
7500
7550
7600
7650
7700
7750
7800
7850
7900
7950
8000

0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

781
794
804
819
828
840
851
859
867
879
887
899
903
912
923
933
942
950
960
975
987
997
1008
1016
1032
1043
1054
1060
1068
1078
1087
1099
1108
1120
1134
1147
1154
1163
1172
1180
1190
1203
1213
1226
1232
1240
1250
1259
1267
1277
1286
1295
1306
1316
1323
1335
1346
1359
1371
1381
1390
1401
1411
1421
1426
1438
1451
1462
1474
1481
1489
1499
1511
1528
1535
1547
1559
1567
1579
1588

285
288
291
296
298
301
302
305
309
312
313
318
320
322
325
328
331
339
340
342
344
348
351
353
354
357
363
367
369
371
376
380
380
382
387
392
395
400
403
405
410
412
416
418
420
424
426
427
429
429
431
438
439
441
442
446
450
452
455
457
460
466
470
473
475
477
480
481
486
491
494
497
501
503
506
507
511
516
519
524

0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

0.192839506
0.193658537
0.19373494
0.195
0.194823529
0.195348837
0.195632184
0.195227273
0.194831461
0.195333333
0.194945055
0.195434783
0.194193548
0.194042553
0.194315789
0.194375
0.194226804
0.193877551
0.193939394
0.195
0.195445545
0.195490196
0.195728155
0.195384615
0.196571429
0.196792453
0.197009346
0.196296296
0.195963303
0.196
0.195855856
0.19625
0.196106195
0.196491228
0.197217391
0.197758621
0.197264957
0.197118644
0.19697479
0.196666667
0.196694215
0.197213115
0.197235772
0.197741935
0.19712
0.196825397
0.196850394
0.19671875
0.196434109
0.196461538
0.196335878
0.196212121
0.196390977
0.19641791
0.196
0.196323529
0.19649635
0.196956522
0.197266187
0.197285714
0.197163121
0.197323944
0.197342657
0.197361111
0.196689655
0.196986301
0.197414966
0.197567568
0.197852349
0.197466667
0.197218543
0.197236842
0.19751634
0.198441558
0.198064516
0.198333333
0.198598726
0.19835443
0.198616352
0.1985

0.07037037
0.070243902
0.070120482
0.07047619
0.070117647
0.07
0.069425287
0.069318182
0.069438202
0.069333333
0.068791209
0.069130435
0.068817204
0.068510638
0.068421053
0.068333333
0.068247423
0.069183673
0.068686869
0.0684
0.068118812
0.068235294
0.06815534
0.067884615
0.067428571
0.067358491
0.067850467
0.067962963
0.067706422
0.067454545
0.067747748
0.067857143
0.067256637
0.067017544
0.067304348
0.067586207
0.067521368
0.06779661
0.067731092
0.0675
0.067768595
0.067540984
0.067642276
0.067419355
0.0672
0.067301587
0.067086614
0.06671875
0.066511628
0.066
0.065801527
0.066363636
0.066015038
0.065820896
0.065481481
0.065588235
0.065693431
0.065507246
0.065467626
0.065285714
0.065248227
0.065633803
0.065734266
0.065694444
0.065517241
0.065342466
0.065306122
0.065
0.065234899
0.065466667
0.065430464
0.065394737
0.065490196
0.065324675
0.065290323
0.065
0.065095541
0.065316456
0.065283019
0.0655

0.367407407
0.36804878
0.36746988
0.369047619
0.369411765
0.370232558
0.369885057
0.369090909
0.368764045
0.369333333
0.369230769
0.37
0.369892473
0.369574468
0.369473684
0.369583333
0.369484536
0.369795918
0.369292929
0.3694
0.369306931
0.370196078
0.370679612
0.370192308
0.370666667
0.370566038
0.371588785
0.370740741
0.371559633
0.370909091
0.370810811
0.371785714
0.371327434
0.37122807
0.371826087
0.372586207
0.372478632
0.372881356
0.372941176
0.372333333
0.372231405
0.372459016
0.37203252
0.372258065
0.37136
0.371746032
0.371181102
0.3709375
0.370852713
0.370461538
0.370229008
0.370757576
0.371578947
0.371343284
0.370518519
0.370735294
0.371386861
0.371594203
0.371654676
0.371428571
0.370780142
0.370704225
0.37048951
0.370277778
0.370206897
0.369863014
0.37047619
0.37027027
0.370469799
0.370533333
0.370066225
0.370131579
0.370326797
0.371038961
0.370709677
0.370769231
0.371210191
0.37164557
0.371572327
0.3715

422
427
430
435
444
451
456
460
465
471
480
485
497
503
507
513
519
523
528
530
534
543
550
556
560
564
571
575
588
591
595
603
610
614
617
622
630
637
644
649
652
657
659
664
669
678
681
688
696
702
708
714
726
731
736
740
748
753
757
762
764
765
768
772
783
785
792
797
800
807
811
817
821
826
832
838
844
853
856
860

0.104197531
0.104146341
0.103614458
0.103571429
0.104470588
0.104883721
0.104827586
0.104545455
0.104494382
0.104666667
0.105494505
0.105434783
0.10688172
0.107021277
0.106736842
0.106875
0.107010309
0.106734694
0.106666667
0.106
0.105742574
0.106470588
0.106796117
0.106923077
0.106666667
0.106415094
0.106728972
0.106481481
0.107889908
0.107454545
0.107207207
0.107678571
0.107964602
0.107719298
0.107304348
0.107241379
0.107692308
0.107966102
0.108235294
0.108166667
0.107768595
0.107704918
0.107154472
0.107096774
0.10704
0.107619048
0.107244094
0.1075
0.107906977
0.108
0.108091603
0.108181818
0.109172932
0.109104478
0.109037037
0.108823529
0.10919708
0.109130435
0.108920863
0.108857143
0.108368794
0.107746479
0.107412587
0.107222222
0.108
0.107534247
0.107755102
0.107702703
0.10738255
0.1076
0.107417219
0.1075
0.107320261
0.107272727
0.107354839
0.107435897
0.107515924
0.107974684
0.107672956
0.1075


Table B.4: Simulation Results for the single mission condition for the Pressure Tank System: (Simulations 8050-10000)

<table>
<thead>
<tr>
<th>Simulation Number</th>
<th>Number of failures</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
<th>Phase 4</th>
<th>Unreliability</th>
<th>Mission Unreliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>8050</td>
<td>0</td>
<td>1599</td>
<td>865</td>
<td>530</td>
<td>0</td>
<td>0.19863554</td>
<td>0.107453416</td>
</tr>
<tr>
<td>8100</td>
<td>0</td>
<td>1613</td>
<td>870</td>
<td>534</td>
<td>0</td>
<td>0.199135802</td>
<td>0.107407407</td>
</tr>
<tr>
<td>8150</td>
<td>0</td>
<td>1622</td>
<td>876</td>
<td>537</td>
<td>0</td>
<td>0.199018405</td>
<td>0.107484663</td>
</tr>
<tr>
<td>8200</td>
<td>0</td>
<td>1628</td>
<td>884</td>
<td>540</td>
<td>0</td>
<td>0.198536585</td>
<td>0.107804878</td>
</tr>
<tr>
<td>8250</td>
<td>0</td>
<td>1637</td>
<td>887</td>
<td>542</td>
<td>0</td>
<td>0.198424242</td>
<td>0.107514451</td>
</tr>
<tr>
<td>8300</td>
<td>0</td>
<td>1649</td>
<td>896</td>
<td>543</td>
<td>0</td>
<td>0.19864999</td>
<td>0.107514451</td>
</tr>
<tr>
<td>8350</td>
<td>0</td>
<td>1658</td>
<td>901</td>
<td>547</td>
<td>0</td>
<td>0.198528747</td>
<td>0.107951807</td>
</tr>
<tr>
<td>8400</td>
<td>0</td>
<td>1668</td>
<td>906</td>
<td>549</td>
<td>0</td>
<td>0.19871429</td>
<td>0.107857143</td>
</tr>
<tr>
<td>8450</td>
<td>0</td>
<td>1685</td>
<td>909</td>
<td>551</td>
<td>0</td>
<td>0.199408284</td>
<td>0.107577964</td>
</tr>
<tr>
<td>8500</td>
<td>0</td>
<td>1697</td>
<td>913</td>
<td>555</td>
<td>0</td>
<td>0.199647059</td>
<td>0.107176166</td>
</tr>
<tr>
<td>8550</td>
<td>0</td>
<td>1705</td>
<td>917</td>
<td>557</td>
<td>0</td>
<td>0.19915205</td>
<td>0.107514451</td>
</tr>
<tr>
<td>8600</td>
<td>0</td>
<td>1709</td>
<td>924</td>
<td>562</td>
<td>0</td>
<td>0.19872093</td>
<td>0.107418666</td>
</tr>
<tr>
<td>8650</td>
<td>0</td>
<td>1722</td>
<td>930</td>
<td>564</td>
<td>0</td>
<td>0.199075145</td>
<td>0.107514451</td>
</tr>
<tr>
<td>8700</td>
<td>0</td>
<td>1732</td>
<td>934</td>
<td>565</td>
<td>0</td>
<td>0.1990846</td>
<td>0.107563222</td>
</tr>
<tr>
<td>8750</td>
<td>0</td>
<td>1744</td>
<td>940</td>
<td>569</td>
<td>0</td>
<td>0.19934286</td>
<td>0.107428571</td>
</tr>
<tr>
<td>8800</td>
<td>0</td>
<td>1752</td>
<td>948</td>
<td>571</td>
<td>0</td>
<td>0.199090909</td>
<td>0.107727273</td>
</tr>
<tr>
<td>8850</td>
<td>0</td>
<td>1760</td>
<td>957</td>
<td>574</td>
<td>0</td>
<td>0.198870056</td>
<td>0.108135593</td>
</tr>
<tr>
<td>8900</td>
<td>0</td>
<td>1768</td>
<td>963</td>
<td>576</td>
<td>0</td>
<td>0.198651685</td>
<td>0.108202247</td>
</tr>
<tr>
<td>8950</td>
<td>0</td>
<td>1780</td>
<td>968</td>
<td>578</td>
<td>0</td>
<td>0.198828082</td>
<td>0.108156425</td>
</tr>
<tr>
<td>9000</td>
<td>0</td>
<td>1789</td>
<td>977</td>
<td>581</td>
<td>0</td>
<td>0.198777778</td>
<td>0.108555556</td>
</tr>
<tr>
<td>9050</td>
<td>0</td>
<td>1798</td>
<td>979</td>
<td>586</td>
<td>0</td>
<td>0.198674033</td>
<td>0.108471910</td>
</tr>
<tr>
<td>9100</td>
<td>0</td>
<td>1805</td>
<td>983</td>
<td>587</td>
<td>0</td>
<td>0.198516484</td>
<td>0.108021978</td>
</tr>
<tr>
<td>9150</td>
<td>0</td>
<td>1811</td>
<td>991</td>
<td>590</td>
<td>0</td>
<td>0.197923497</td>
<td>0.108306011</td>
</tr>
<tr>
<td>9200</td>
<td>0</td>
<td>1819</td>
<td>994</td>
<td>594</td>
<td>0</td>
<td>0.197737391</td>
<td>0.108437478</td>
</tr>
<tr>
<td>9250</td>
<td>0</td>
<td>1829</td>
<td>999</td>
<td>597</td>
<td>0</td>
<td>0.19772973</td>
<td>0.108454051</td>
</tr>
<tr>
<td>9300</td>
<td>0</td>
<td>1841</td>
<td>1004</td>
<td>601</td>
<td>0</td>
<td>0.197569889</td>
<td>0.107950899</td>
</tr>
<tr>
<td>9350</td>
<td>0</td>
<td>1851</td>
<td>1012</td>
<td>603</td>
<td>0</td>
<td>0.197967114</td>
<td>0.108235294</td>
</tr>
<tr>
<td>9400</td>
<td>0</td>
<td>1861</td>
<td>1018</td>
<td>608</td>
<td>0</td>
<td>0.197978723</td>
<td>0.108297872</td>
</tr>
<tr>
<td>9450</td>
<td>0</td>
<td>1874</td>
<td>1018</td>
<td>610</td>
<td>0</td>
<td>0.198368378</td>
<td>0.107724868</td>
</tr>
<tr>
<td>9500</td>
<td>0</td>
<td>1884</td>
<td>1021</td>
<td>612</td>
<td>0</td>
<td>0.198315789</td>
<td>0.107473684</td>
</tr>
<tr>
<td>9550</td>
<td>0</td>
<td>1895</td>
<td>1027</td>
<td>613</td>
<td>0</td>
<td>0.198429319</td>
<td>0.107593267</td>
</tr>
<tr>
<td>9600</td>
<td>0</td>
<td>1907</td>
<td>1032</td>
<td>616</td>
<td>0</td>
<td>0.198645833</td>
<td>0.107510768</td>
</tr>
<tr>
<td>9650</td>
<td>0</td>
<td>1914</td>
<td>1037</td>
<td>619</td>
<td>0</td>
<td>0.198349691</td>
<td>0.107461114</td>
</tr>
<tr>
<td>9700</td>
<td>0</td>
<td>1923</td>
<td>1043</td>
<td>623</td>
<td>0</td>
<td>0.198247423</td>
<td>0.107525773</td>
</tr>
<tr>
<td>9750</td>
<td>0</td>
<td>1933</td>
<td>1051</td>
<td>625</td>
<td>0</td>
<td>0.19825641</td>
<td>0.107794872</td>
</tr>
<tr>
<td>9800</td>
<td>0</td>
<td>1948</td>
<td>1059</td>
<td>630</td>
<td>0</td>
<td>0.1977551</td>
<td>0.108061224</td>
</tr>
<tr>
<td>9850</td>
<td>0</td>
<td>1956</td>
<td>1063</td>
<td>634</td>
<td>0</td>
<td>0.1985768</td>
<td>0.107918782</td>
</tr>
<tr>
<td>9900</td>
<td>0</td>
<td>1971</td>
<td>1067</td>
<td>637</td>
<td>0</td>
<td>0.199090909</td>
<td>0.107777778</td>
</tr>
<tr>
<td>9950</td>
<td>0</td>
<td>1985</td>
<td>1073</td>
<td>643</td>
<td>0</td>
<td>0.19947487</td>
<td>0.107839196</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
<td>1990</td>
<td>1080</td>
<td>645</td>
<td>0</td>
<td>0.199</td>
<td>0.108</td>
</tr>
</tbody>
</table>
Table B.5: Analytical Results for the multiple mission condition for the Pressure Tank System

<table>
<thead>
<tr>
<th>Number of cycles</th>
<th>Simulation Time</th>
<th>Number of failures</th>
<th>Phase Unreliability</th>
<th>Mission Unreliability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Phase 1</td>
<td>Phase 2 Phase 3</td>
<td>Phase 4 Phase 1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2 0</td>
<td>0 0.0000</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>9</td>
<td>17 4</td>
<td>1 0.0018</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>23</td>
<td>29 7</td>
<td>2 0.0046</td>
</tr>
<tr>
<td>20</td>
<td>60</td>
<td>64</td>
<td>56 14</td>
<td>5 0.0128</td>
</tr>
<tr>
<td>30</td>
<td>90</td>
<td>94</td>
<td>107 29</td>
<td>17 0.0188</td>
</tr>
<tr>
<td>40</td>
<td>120</td>
<td>123</td>
<td>131 32</td>
<td>18 0.0246</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
<td>168</td>
<td>154 46</td>
<td>36 0.0336</td>
</tr>
<tr>
<td>100</td>
<td>300</td>
<td>298</td>
<td>328 73</td>
<td>48 0.0596</td>
</tr>
<tr>
<td>200</td>
<td>600</td>
<td>591</td>
<td>594 138</td>
<td>84 0.1182</td>
</tr>
<tr>
<td>300</td>
<td>900</td>
<td>819</td>
<td>840 201</td>
<td>125 0.1638</td>
</tr>
<tr>
<td>400</td>
<td>1200</td>
<td>1040</td>
<td>993 233</td>
<td>132 0.2080</td>
</tr>
<tr>
<td>500</td>
<td>1500</td>
<td>1215</td>
<td>1183 275</td>
<td>156 0.2430</td>
</tr>
<tr>
<td>1000</td>
<td>3000</td>
<td>1686</td>
<td>1712 382</td>
<td>229 0.3372</td>
</tr>
<tr>
<td>2000</td>
<td>6000</td>
<td>2007</td>
<td>2057 448</td>
<td>284 0.4014</td>
</tr>
<tr>
<td>3000</td>
<td>9000</td>
<td>2082</td>
<td>2130 453</td>
<td>281 0.4164</td>
</tr>
<tr>
<td>4000</td>
<td>12000</td>
<td>2111</td>
<td>2124 475</td>
<td>271 0.4222</td>
</tr>
<tr>
<td>5000</td>
<td>15000</td>
<td>2123</td>
<td>2126 473</td>
<td>264 0.4246</td>
</tr>
</tbody>
</table>
C.1 Input Files

This section includes the files used to generate the Petri net based on the repairable bulb system.

C.1.1 Project File

- - PROJECT bulb_system

- - DT files:

toggleSwitch.dt;
powerSupply.dt;
b_operator.dt;
bulb.dt;
junctionOneIn.dt;
junctionTwoIn.dt;

- - OMT files:

toggleSwitch.omt;

- - SS file:

ss_b1.ss

- - PTT file:

ptt_b1.ptt;

- - SIM file:

simulation_b1.sim;
C.1.2 Component Files

This section covers the decision and operational mode table files of each component type within the bulb system.

C.1.2.1 Component Power Supply Decision Table

File Name: powerSupply.dt

DT power_supply
{
in:in1, state:state1, out:out1;
C, W, C;
-, F, NC;
NC, -, NC;
-, UR, NC;
}

C.1.2.2 Component Toggle Switch Decision and Operational Mode Table

File Name: toggleSwitch.dt

DT toggle_switch
{
in:in2, mode:mode3, out:out1;
-, open, NC;
NC, -, NC;
C, closed, C;
}

OMT toggle_switch
{
mode:mode1, in:in1, state:state1, mode:mode2;
closed, -, F_closed, closed;
closed, CL, -, closed;
closed, OP, W, open;
closed, NA, -, closed;
closed, -, UR, open;
open, -, F_open, open;
open, OP, -, open;
open, NA, -, open;
open, CL, W, closed;
open, -, UR, open;
}

C.1.2.3 Component Operator Decision Table

File Name: b_operator.dt

DT_operator
{
time:time1, in:in1, state:state1, out:out1;
0, OFF, W, CL;
(0<t<=20), OFF, W, NA;
(0<t<20), ON, W, NA;
20, ON, W, OP;
-, -, F, NA;
}

C.1.2.4 Component Bulb Decision Table

File Name: bulb.dt

DT bulb
{
in:in1, state:state1, out:out1, out:out2;
C, W, C, ON;
NC, -, NC, OFF;
-, F, NC, OFF;
-, UR, NC, OFF;
}

C.1.2.5 Component Junction Decision Tables

File Name: junctionTwoIn.dt

DT junction_two_in
{
in:in1, in:in2, out:out1;
C, -, C;
Appendix C. Bulb System

File Name: junctionOneIn.dt

DT junction_one_in
{
in:in1, out:out1, out:out2;
C, C, C;
NC, NC, NC;
}

C.1.3 System Structure File

File Name: ss_b1.ss

SS SS_Structure_bulb

begin
OP : b_operator
port map(
in1 => B_OP;
out1 => OP_S1;
)

S1 : toggle_switch
port map(
in1 => OP_S1;
in2 => J2_S1;
out1 => S1_B;
)

PS1 : power_supply
port map(
in1 => J1_PS1;
out1 => PS1_J2;
)
PS2 : power_supply
port map(
in1 => J1_PS2;
out1 => PS2_J2;
)

J1 : junction_one_in
port map(
in1 => B_J1;
out1 => J1_PS1;
out2 => J1_PS2;
)

J2 : junction_two_in
port map(
in1 => PS1_J2;
in2 => PS2_J2;
out1 => J2_S1;
)

B : bulb
port map(
in1 => S1_B;
out1 => B_J1;
out2 => B_OP;
)
end

C.1.4 Phase Transition Table File

File Name: ppt_b1.ptt

ptt mission_b1
{
time, from_phase, to_phase, condition;
0, 1, 2, B.out2=ON;
delta, 1, 4, B.out2=OFF;
20, 2, 3, B.out2=OFF;
-, 2, 5, B.out2=OFF;
delta, 2, 6, B.out2=ON;
}

C.1.5 Simulation File

File Name: simulation_b1.sim

SIM
COMPONENT MODES
{
S1 mode = open;
}

FAILURE
{
S1 F_closed exponential(1000);
S1 F_open exponential(1000);
PS1 F exponential(150.00);
B F exponential(200.00);
PS2 F exponential(150.00);
}
REPAIR
{
S1 exponential(10);
PS1 exponential(50.000);
PS2 exponential(50.000);
B exponential(10.000);
}
-- Maintenance Plan
MAINTENANCE
{ corrective PS1(eng1);
corrective PS2(eng2);
corrective S1(eng4);
corrective B(eng5);
C.1. Input Files

}\n- -STANDBY
{
  PS1 * COLD(PS2);
  PS2 COLD(PS1);
}\n
INITIAL
{
  OP out1 = CL;
}\n
C.1.6 Setup File

File Name: setup.ini

- SETUP FILE -
INI
- - DEFAULT
- - Working state of components

  working state :: W;

- - Default under repair state
  repair state :: UR;
- - Wire link types current, no current used for the decision tables
- - List as Current, noCurrent

  wire type :: C, NC;