Formal Specification and Runtime Verification of Parallel Systems using Interval Temporal Logic (ITL)

PhD Thesis

Nayef Hmoud Alshammari

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Software Technology Research Laboratory
De Montfort University
Leicester - United Kingdom

2018
Declaration of Authorship

I declare that the work described in this thesis is original work undertaken by me for the degree of Doctor of Philosophy at the Software Technology Research Laboratory (STRL), at De Montfort University, United Kingdom.

No part of the material described in this thesis has been submitted for any award of any other degree or qualification in this or any other university or college of advanced education. This thesis is written by me and produced using \LaTeX.
To my mother’s soul, Helalah bint Huja'lan Alshammari (April 21st, 2016).

May she rest in peace ...
Acknowledgement

Firstly, I would like to express my sincere gratitude to my first supervisor Dr. Francois Seiwe and second supervisor Dr. Antonio Cau for their continuous support of my PhD study and related research, and for their patience, motivation, and immense knowledge. Their guidance helped me during the time of my research and writing of this thesis. I could not have had more qualified supervisors for my PhD study.

Also, I would like to express my great appreciation to Dr. Antonio Cau for the remarkable theoretical and practical support which he offered during the study of my PhD. I would also like to thank my former supervisors Dr. Ben Moszkowski and Dr. Jordan Dimitrov for their roles during my PhD.

Special deep thanks and appreciation to those who have endlessly and continuously believed in me and supported me especially my great sisters, Loulouah and Khaznah, and my best friend Dr. Dheidan Alshammari. Big thanks to my family, my friends and all those who have been relentlessly supportive during my PhD journey.
Abstract

Runtime Verification (RV) is the discipline that allows monitoring systems at runtime in order to check the satisfaction or violation of a given correctness property. Parallel systems are more complicated than sequential systems. Therefore, systems that run in parallel need a parallel runtime verification framework to monitor their behaviour and guarantee correctness properties. Parallel systems have correctness properties different from correctness properties of sequential systems. For instance, as a correctness property of parallel systems, absence of deadlock has to be guaranteed and mutual exclusion mechanism has to be applied in case a resource is shared between more than one system and the parallelism form is true concurrency. Therefore, sequential runtime verification framework can not handle systems that run in parallel due to the singularity issue of this kind of framework as they are built to handle a single system at a time, whereas for parallel systems a framework has to handle many systems at a time. AnaTempura is a runtime verification tool which can handle single systems at a time. To solve this problem, I evolved AnaTempura to be able to handle parallel systems. In this thesis, I propose a Parallel Runtime Verification Framework (P3VF) that can handle systems which use architectures of parallelism in their design such as multi-core processor architecture. The proposed model can check system behaviour at runtime in order to either guarantee satisfaction or detect violations of correctness properties. My technique is based on Interval Temporal Logic (ITL) and its executable subset Tempura to verify properties at runtime using the AnaTempura tool.

I use, as a demonstration, the case study of private L2 cache memory of multi-core processor architecture. My objectives are to i) design MSI protocol compliant with cache memory
coherence and \(ii\) fulfil main memory consistency model at runtime. I achieve this via a formal Tempura specification of the cache controller which is then verified at runtime against my objectives for memory consistency and cache coherence using AnaTempura. The presented specifications allow to extend it to not only capture correctness but also monitor the performance of a cache memory controller. The case study is then evaluated via integrating AnaTempura with MATLAB in order to check correctness properties such as memory consistency and cache coherence.
Contents

Declaration of Authorship I

Acknowledgments III

Abstract IV

List of Figures XII

List of Tables XVII

List of Abbreviations XVIII

1 Introduction 1
  1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.2 Problem Statement and Research Motivation . . . . . . . . . . . . . . . . . . . 3
  1.3 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  1.4 Research Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  1.5 Success Criteria . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  1.6 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2 Verification Techniques for Parallel Systems: a Review 11
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.2 Basic Concepts and Related Topics . . . . . . . . . . . . . . . . . . . . . . . . 12
## CONTENTS

2.6.2.1 Syntax ................................................................. 34
2.6.2.2 Informal Semantics ............................................... 35
2.6.2.3 Justification for Choosing Interval Temporal Logic (ITL) ... 36

2.7 Related Work ............................................................ 37
  2.7.1 Memory Models for Interval Temporal Logic (ITL) ............. 37
    2.7.1.1 Framing Variables ........................................... 37
    2.7.1.2 Transactional Memory ...................................... 38
  2.7.2 Meltdown and Spectre ............................................. 40

2.8 Summary ............................................................... 41

3 Computational Model .................................................... 42
  3.1 Introduction .......................................................... 43
  3.2 Computational Model ................................................ 43
    3.2.1 Message-Passing based Communication ......................... 43
      3.2.1.1 Related Work ............................................... 44
      3.2.1.2 Execution Modes ............................................ 45
      3.2.1.3 Channel Communication .................................... 45
      3.2.1.4 Shunt Communication ...................................... 47
      3.2.1.5 Delay and Timeout ......................................... 48
      3.2.1.6 Resource Allocation ....................................... 48
      3.2.1.7 The Funnel .................................................. 48
    3.2.2 Shared-Variable based Communication .......................... 49
    3.2.3 True Concurrency ............................................... 50
    3.2.4 Interleaving Concurrency ...................................... 51

3.3 Architecture Framework .............................................. 53
  3.3.1 Generation Phase ................................................ 53
## CONTENTS

3.3.1.1 Communication Models ........................................ 54
3.3.1.2 Concurrency Forms ........................................... 54
3.3.1.3 Execution Modes ............................................. 54

3.3.2 Locals Verification & Assertion Phase ........................... 56
3.3.2.1 Interleaving Concurrency and Shared-Variable .............. 56
3.3.2.2 True Concurrency and Shared-Variable ...................... 58
3.3.2.3 Synchronous Execution and Message-Passing (Channels) ... 60
3.3.2.4 Asynchronous Execution and Message-Passing (Shunts) .... 62

3.3.3 Global Verification Phase ........................................ 65

3.4 Parallel Runtime Verification Framework (PRVF) Model ........... 66

3.5 Summary .......................................................... 66

4 Design and Implementation of a Parallel Runtime Verification Framework (PRVF) 68
4.1 Introduction ....................................................... 69
4.2 (Ana)Tempura .................................................... 69
4.2.1 Assertion Points ................................................ 71
4.2.2 The Monitor .................................................... 76
4.2.3 Tempura Interpreter ........................................... 77

4.3 Evolutionary Improvements of AnaTempura ......................... 77
4.3.1 Realisation of Assertion Points Techniques .................... 80
4.4 Benchmarking Applications ........................................ 90
4.4.1 Producer-Consumer ............................................ 90
4.4.2 Dining Philosophers Problem .................................. 92
4.5 Summary .......................................................... 93

5 Case Study: Cache Controller ........................................... 94
5.1 Cache Memory Controller: A Case Study ........................... 95
## CONTENTS

5.2 The Basics of Cache Memory ............................................. 95  
5.2.1 Description ......................................................... 97  
5.2.2 MSI Protocol ........................................................ 101  
5.2.3 Formal Description of Cache Controller ....................... 102  
5.2.4 Compositional Modelling .......................................... 103  
5.3 Analysis and Discussion ............................................... 111  
5.3.1 Global Program : Cache Controller ............................ 113  
5.3.1.1 Raw Data Description ........................................ 113  
5.3.1.2 External Programs : Local Processors .................... 118  
5.3.1.3 Raw Data Analysis ............................................ 120  
5.3.1.4 Properties Check of The Cache Controller ............... 127  
5.4 Summary .............................................................. 128  

6 Evaluation of Parallel Runtime Verification Framework (PRVF) 129  
6.1 Introduction ............................................................ 130  
6.2 MATLAB ............................................................... 131  
6.3 Integrating MATLAB and AnaTempura ............................ 132  
6.3.1 Running MATLAB ............................................... 132  
6.3.2 AnaTempura Runs MATLAB .................................... 135  
6.4 Correctness Properties ............................................... 140  
6.4.1 Revisiting The Case Study of Cache Controller ............ 140  
6.4.2 Memory Consistency Property .................................. 144  
6.4.3 Cache Coherence Property ....................................... 156  
6.5 Discussion ............................................................. 167  
6.6 Related Work .......................................................... 168  
6.7 Summary .............................................................. 172
## CONTENTS

### 7 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Thesis Summary</td>
<td>174</td>
</tr>
<tr>
<td>7.2 Comparison with Related Work</td>
<td>176</td>
</tr>
<tr>
<td>7.3 Original Contribution</td>
<td>177</td>
</tr>
<tr>
<td>7.4 Success Criteria Revisited</td>
<td>178</td>
</tr>
<tr>
<td>7.5 Limitations</td>
<td>180</td>
</tr>
<tr>
<td>7.6 Future Work</td>
<td>181</td>
</tr>
<tr>
<td>7.7 Future Impact</td>
<td>183</td>
</tr>
<tr>
<td>7.7.1 Academic</td>
<td>183</td>
</tr>
<tr>
<td>7.7.2 Industrial</td>
<td>183</td>
</tr>
</tbody>
</table>

### Bibliography

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Appendix A: Simulations &amp; Animation</td>
<td>216</td>
</tr>
<tr>
<td>B Appendix B: Tempura Code for Cache Controller</td>
<td>227</td>
</tr>
<tr>
<td>C Appendix C: Tcl/tk Code for Cache Controller</td>
<td>282</td>
</tr>
<tr>
<td>D Appendix D: Java Remote Method Invocation (RMI)</td>
<td>317</td>
</tr>
<tr>
<td>E Appendix E: MATLAB Code for Correctness Properties</td>
<td>326</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Modelling Parallelism using Petri Net [200] ........................................... 16
2.2 Verification Techniques ................................................................. 21
2.3 Taxonomy of Runtime Verification [151] ............................................ 24
2.4 Some Sample of ITL Formulae [51] ...................................................... 35
2.5 Chop ............................................................................................... 35
2.6 Chop Star ......................................................................................... 36
3.1 Synchronous Execution of Parallel Systems $Sys_1$ and $Sys_2$ ................. 45
3.2 Asynchronous Execution of Parallel Systems $Sys_1$ and $Sys_2$ ................. 45
3.3 Parallel Composition of $Sys_1$ and $Sys_2$ (True Concurrency) ................. 50
3.4 Global State Construction (True Concurrency) ....................................... 51
3.5 Parallel Composition of $Sys_1$ and $Sys_2$ (Interleaving Concurrency) ....... 52
3.6 Global State Construction (Interleaving Concurrency) ............................ 53
3.7 Parallel Runtime Verification Framework (Shared-Variable Interleaving Concurrency) ................................................................. 57
3.8 Parallel Runtime Verification Framework (Shared-Variable True Concurrency) ...... 59
3.9 Parallel Runtime Verification Framework (Synchronous Message-Passing) ..... 61
3.10 Parallel Runtime Verification Framework (Asynchronous Message-Passing) .... 64
4.1 General System Architecture of AnaTempura [301] ................................. 69
4.2 The Analysis Process [301] ................................................................. 70
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Assertion Points and Chunks</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Processing Assertion Points</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Compiling External Java Program</td>
<td>74</td>
</tr>
<tr>
<td>4.6</td>
<td>Running Tempura Program</td>
<td>76</td>
</tr>
<tr>
<td>4.7</td>
<td>Amdahl’s Law [115]</td>
<td>78</td>
</tr>
<tr>
<td>4.8</td>
<td>Runtime Verification</td>
<td>79</td>
</tr>
<tr>
<td>4.9</td>
<td>Generating Extended Assertion Points within External Java Program</td>
<td>81</td>
</tr>
<tr>
<td>4.10</td>
<td>Collecting Extended Assertion Points Tempura Program</td>
<td>82</td>
</tr>
<tr>
<td>4.11</td>
<td>Global Collects Assertion Points from Locals Tempura Program</td>
<td>87</td>
</tr>
<tr>
<td>4.12</td>
<td>Implementation Java RMI using AnaTempura</td>
<td>89</td>
</tr>
<tr>
<td>4.13</td>
<td>Producer-Consumer</td>
<td>90</td>
</tr>
<tr>
<td>4.14</td>
<td>Producer-Consumer Execution in Tempura/AnaTempura</td>
<td>91</td>
</tr>
<tr>
<td>4.15</td>
<td>Demo of Dining Philosophers Problem</td>
<td>92</td>
</tr>
<tr>
<td>5.1</td>
<td>Dual Core Dual Processor System</td>
<td>98</td>
</tr>
<tr>
<td>5.2</td>
<td>Cache Controller</td>
<td>112</td>
</tr>
<tr>
<td>5.3</td>
<td>Tempura Execution at State 0</td>
<td>114</td>
</tr>
<tr>
<td>5.4</td>
<td>AnaTempura Simulation at State 0</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Local States &amp; Properties of Processors 0, 1, 2 at State 0</td>
<td>119</td>
</tr>
<tr>
<td>5.6</td>
<td>States &amp; Intervals ($\sigma^m_n$, where $m$ is state number, $n$ is Processor id) of Cache Controller and Memory values</td>
<td>126</td>
</tr>
<tr>
<td>6.1</td>
<td>Running MATLAB</td>
<td>133</td>
</tr>
<tr>
<td>6.2</td>
<td>MATLAB Script</td>
<td>133</td>
</tr>
<tr>
<td>6.3</td>
<td>MATLAB Arithmetic Script Output</td>
<td>134</td>
</tr>
<tr>
<td>6.4</td>
<td>AnaTempura inputs numbers to file “input.txt”</td>
<td>137</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

6.5   File Content for “input.txt” ......................................................... 138
6.6   MATLAB Reads from input file “input.txt” ................................. 139
6.7   AnaTempura Run of L2 Cache Memory of Processors 0, 1 & 2 ......... 141
6.8   Tcl Animation of L2 Cache Memory of Processors 0, 1 & 2 ............. 142
6.9   External Programs of AnaTempura for Processors 4 & 5 ................. 143
6.10  Dual Core Dual Processor System ............................................... 143
6.11  Memory Consistency Check at State 0 .......................................... 146
6.12  Memory Consistency Check at State 1 .......................................... 147
6.13  Memory Consistency Check at State 2 .......................................... 148
6.14  Memory Consistency Check at State 3 .......................................... 149
6.15  Memory Consistency Check at State 4 .......................................... 150
6.16  Memory Consistency Check at State 5 .......................................... 151
6.17  Memory Consistency Check at State 6 .......................................... 152
6.18  Memory Consistency Check at State 7 .......................................... 153
6.19  Memory Consistency Check at State 8 .......................................... 154
6.20  Memory Consistency Check at State 9 .......................................... 155
6.21  Cache Coherence & MSI Protocol Check at State 0 ......................... 157
6.22  Cache Coherence & MSI Protocol Check at State 1 ......................... 158
6.23  Cache Coherence & MSI Protocol Check at State 2 ......................... 159
6.24  Cache Coherence & MSI Protocol Check at State 3 ......................... 160
6.25  Cache Coherence & MSI Protocol Check at State 4 ......................... 161
6.26  Cache Coherence & MSI Protocol Check at State 5 ......................... 162
6.27  Cache Coherence & MSI Protocol Check at State 6 ......................... 163
6.28  Cache Coherence & MSI Protocol Check at State 7 ......................... 164
6.29  Cache Coherence & MSI Protocol Check at State 8 ......................... 165
6.30  Cache Coherence & MSI Protocol Check at State 9 ......................... 166
LIST OF FIGURES

A.1 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 0 . . . . . . . . . . 217
A.2 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 0 . . . . . . . 217
A.3 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 0 . . . . . . 217
A.4 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 1 . . . . . . . . . . 218
A.5 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 1 . . . . . . . 218
A.6 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 1 . . . . . . 218
A.7 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 2 . . . . . . . . . . 219
A.8 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 2 . . . . . . . 219
A.9 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 2 . . . . . . 219
A.10 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 3 . . . . . . . . . . 220
A.11 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 3 . . . . . . . 220
A.12 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 3 . . . . . . 220
A.13 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 4 . . . . . . . . . . 221
A.14 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 4 . . . . . . . 221
A.15 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 4 . . . . . . 221
A.16 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 5 . . . . . . . . . . 222
A.17 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 5 . . . . . . . 222
A.18 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 5 . . . . . . 222
A.19 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 6 . . . . . . . . . . 223
A.20 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 6 . . . . . . . 223
A.21 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 6 . . . . . . 223
A.22 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 7 . . . . . . . . . . 224
A.23 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 7 . . . . . . . 224
A.24 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 7 . . . . . . 224
A.25 CACHÉ CONTROLLER EXECUTION IN TEMPURA AT STATE 8 . . . . . . . . . . 225
A.26 CACHÉ CONTROLLER SIMULATION IN ANATEMPURA AT STATE 8 . . . . . . . 225

XV
LIST OF FIGURES

A.27 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 8 . . . . . . 225
A.28 CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 9 . . . . . . . . . . 226
A.29 CACHE CONTROLLER SIMULATION IN ANATEMPURA AT STATE 9 . . . . . . 226
A.30 LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 9 . . . . . 226
## List of Tables

2.1 Intel vs AMD Processors ........................................... 15  
2.2 LTL vs CTL vs ITL .................................................. 33  
2.3 Syntax of ITL ....................................................... 34  
5.1 9-Memory Refernces to 8-Blocks Cache .......................... 99  
5.2 Empty 8-Blocks Cache ............................................. 100  
5.3 Miss of Address [10110] ............................................ 100  
5.4 Miss of Address [11010] .......................................... 100  
5.5 Miss of Address [10000] .......................................... 100  
5.6 Miss of Address [00011] .......................................... 101  
5.7 Miss of Address [10010] .......................................... 101  
5.8 MSI Protocol ....................................................... 102  
5.9 TEMPURA SYNTAX VERSUS ITL SYNTAX .......................... 108  
5.10 TEMPURA RUN OF INTERLEAVED PARALLEL LOCAL PROCESSORS 0, 1 & 2 . 117  
5.11 REQUESTS OF PID$_0$, PID$_1$, & PID$_2$ RESPECTIVELY. .................. 125  
5.12 L2 CACHE MEMORY OF PID$_0$, PID$_1$, & PID$_2$ RESPECTIVELY. .......... 125  
5.13 PROPERTIES CHECK OF CACHES OF PROCESSORS 0, 1 & 2 ............... 127  
6.1 MSI Protocol ....................................................... 156  
6.2 Parallel Computational Models ................................... 171
# List of Abbreviation

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITL</td>
<td>Interval Temporal Logic</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>CTL</td>
<td>Computational Tree Logic</td>
</tr>
<tr>
<td>PTL</td>
<td>Projection Temporal Logic</td>
</tr>
<tr>
<td>TM</td>
<td>Transactional Memory</td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity Consistency Isolation Durability</td>
</tr>
<tr>
<td>Mutex</td>
<td>Mutual Exclusion</td>
</tr>
<tr>
<td>PRVF</td>
<td>Parallel Runtime Verification Framework</td>
</tr>
<tr>
<td>TAM</td>
<td>Temporal Agent Model</td>
</tr>
<tr>
<td>wtr</td>
<td>willing to read</td>
</tr>
<tr>
<td>wtw</td>
<td>willing to write</td>
</tr>
<tr>
<td>Var</td>
<td>Variable</td>
</tr>
<tr>
<td>Val</td>
<td>Value</td>
</tr>
<tr>
<td>Addr</td>
<td>Address</td>
</tr>
<tr>
<td>RW</td>
<td>Read Write</td>
</tr>
<tr>
<td>Pid</td>
<td>Processor identification</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Machine</td>
</tr>
<tr>
<td>PRAM</td>
<td>Parallel Random Access Machine</td>
</tr>
<tr>
<td>EREW</td>
<td>Exclusive Read Exclusive Write</td>
</tr>
<tr>
<td>CREW</td>
<td>Concurrent Read Exclusive Write</td>
</tr>
<tr>
<td>CRCW</td>
<td>Concurrent Read Concurrent Write</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Level 2 Cache</td>
</tr>
<tr>
<td>MSI</td>
<td>Modified Shared Invalid</td>
</tr>
<tr>
<td>MESI</td>
<td>Modified Exclusive Shared Invalid</td>
</tr>
<tr>
<td>MOESI</td>
<td>Modified Owned Exclusive Shared Invalid</td>
</tr>
<tr>
<td>Sys</td>
<td>System</td>
</tr>
<tr>
<td>fin</td>
<td>final</td>
</tr>
<tr>
<td>As/Co</td>
<td>Assumption/Commitment</td>
</tr>
<tr>
<td>DCS</td>
<td>Distributed Control Systems</td>
</tr>
<tr>
<td>OREDA</td>
<td>Offshore and Onshore Reliability Data</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Objectives:

• To set a background for the conducted research

• To identify the problem statement and research motivation

• To raise the research questions

• To provide the research methodology

• To highlight the success criteria

• To provide the thesis outline
CHAPTER 1. INTRODUCTION

1.1 Background

Parallel computing includes computer architecture, operating systems, programming languages, applications, and algorithms. The design and the implementation of these instances have to consider parallelism in order to deliver highly efficient parallel computations. The main goals of parallel computations are improving the speed needed to accomplish tasks and easing the functionality of the tasks being computed. These goals are sometimes difficult to achieve due to hardware or software issues. The key driver of hardware parallelism is the performance of computer systems, while the key drivers of software parallelism are performance and application functionality [211].

Parallel programming is an important factor towards effective parallel computing. The major purpose of parallel programming is the efficient execution of codes in order to save the time needed to execute applications. The efficient execution of codes enables parallel programming to scale well with the problem size, which, consequently, leads to solving larger problems efficiently. This efficient performance of parallel programming is due to providing concurrency which allows simultaneous performing of multiple tasks [210].

Parallel programming goes beyond the limits caused by sequential computing such as physical and practical factors that limit the ability of constructing faster sequential computers [210]. For instance, sequential computers speed is subjected to the speed of data which moves through the hardware. A bandwidth of such medium restricts the transmission through a physical medium (e.g. the speed of light or the transmission limit of copper wire). The technology of semiconductors and evolutionary advancement allow a single chip to have a larger number of transistors; however, reducing the size of such transistors to a molecular or atomic level eventually reaches a limit. Another physical factor which limits sequential computing from being efficient is the processor heat caused by to the amount of the consumed power; thus, dissipating processor heat by conventional way is hard. The development of a faster processor to solve a single computational
problem is increasingly expensive. Therefore, using large number of processors to solve such computational problem is less expensive. The development of parallel systems architecture such as multi-core technology overcomes these kind of problems.

Another reason for developing parallel programs is the use of several computer memory resources instead of using only one computer resource which might be scarce or costly to manage [210]. This advantage of parallel computing, in general, overcomes the limitation of the scarcity of memory resources. The improvement in parallel systems approaches such as multi-core technology has been obtaining continuous gains.

These remarkable benefits make parallel computing the base of future computing systems. As parallel computing systems are becoming ubiquitous in everyday life [142], careful attention has to be paid to the satisfaction of their correctness property.

Verification techniques have to be taken into account during the development of parallel computing systems in order to deliver remarkable benefits of parallel computing. Verification techniques assure a correctness property of parallel computing systems with respect to their specifications. Correctness property is a milestone in systems design and development process. Thus, my research considers verification techniques as a mandatory approach to the correctness of parallel computing systems. Correctness criteria is a preliminary step towards high performance and efficient functionality. In other words, to gain performance it has to pain correctness first.

1.2 Problem Statement and Research Motivation

In this section I will shed light on the research problems. Subsequently, the research gap is addressed and a solution will be proposed. After that, the research motivation is highlighted. I believe the following problems are real research problems and they should be addressed in order to provide suitable solutions which will eventually enhance parallel computing models algorithms and design. The problems are:
• The correctness of parallel programs is harder to determine than for sequential programs [142]. Fixing parallel programs bugs at the software level such as data race, atomicity violation, deadlock are much harder than fixing sequential program ones. On the other hand, at the hardware level a non-determinism execution is a major problem due to the out-of-sync clocks of large systems which cause slight timing variations of a given program. Even though the clock is synced, different interactions with operating systems or other applications could lead to non-deterministic execution of a program each time it runs. The non-deterministic problem makes capturing errors kind of impossible.

• Some parallel programs use synchronisation points to coordinate the work of the overall computations and to ensure that all the parallel operations are synchronised and the data is being used is consistent. Some parallel programs use message-passing approach to exchange data and ensure synchronisation of data being used within the parallel operations of such programs. The latter approach is commonly used in distributed memory machines. However, software developers who use this approach find the correctness of such programs difficult due to the variety of inputs that might be given to these programs. Also, the fact that multiple software developers may share work on a given portion of a program is another difficulty.

• Some parallel programs may use what is so-called critical section (or atomic region) to prevent any access attempt to a shared resource at a time for more than one processor. This mechanism is used to ensure a concurrency characteristic called atomicity. Atomicity violation is considered a concurrency bug. Lock-based algorithm is used to ensure the atomic execution of the critical section (or atomic region) for concurrency sake. Undesired behaviour might occur due to the use of locks which is deadlock problem where two processors are waiting for each other permanently which eventually leads to delay the computation or halt the two processors.
A unified generic model that can handle these concurrency issues with the consideration of different parallel computation model aspects such as concurrency, communication and execution is needed. Therefore, I present in this PhD thesis a unified generic model for parallel computing models considering aspects such as concurrency, communication and execution. Such model will benefit hardware computer systems and software performance and application functionality. Parallel algorithms and analysis is intended to be delivered for the sake of design enhancements and correctness verification at the same time. This model is formal method-based approach which aims to establish an accurate and unambiguous semantics in order to deliver effective description of every phase of parallel systems behaviour in order to fulfil correctness properties such as safety and liveness.

Therefore, correctness of parallel computing is a mandatory need towards achieving the best of parallelism with respect to performance and functionality. Runtime verification plays a major role within verification techniques due to a number of reasons. Some of the most important reasons for using runtime verification over other verification techniques are that it is a lightweight tool, and because it guarantees the absence of states explosion caused by modelling all possible states of system under scrutiny. My approach, runtime verification of parallel computing systems, contributes to move forward the parallelism at hardware design level and software performance and functionality level via the discovery of the behaviour of hardware/software during runtime. In other words, there are states that can not be discovered but at runtime. The monitoring of either satisfaction or violation of correctness criteria is the main task of runtime verification and via this task hardware design and software performance evaluation can be measured accurately. According to Muller [199] the evaluation of verification techniques focuses on the following criteria: Soundness, Completeness, Modularity, Automation, and Efficiency. In my approach I consider these criteria carefully as my aim is to deliver a generic model in order to verify correctness property for parallel systems at runtime.
1.3 Research Questions

The major question of my research is:

- How to infer the correctness of global property from the correctness of a set of local properties for computer systems using parallel architectures?

Out of the raised major research question, a set of research questions are intended to tackle different aspects of the encountered issues. These are:

1. How to construct a global state out of local states of parallel systems?
2. How to compose/decompose a global property out of/into a set of local properties?
3. How to handle different forms of concurrency, communication models, and execution modes of parallel systems?
4. How to verify local/global properties at the local/global level?

1.4 Research Methodology

The research methodology I adopt follows a constructive research approach. My research contributes to knowledge being developed as a new solution for the identified problem. Therefore, I constructed new algorithms, a computational model, and an architecture framework. The construction of this approach consists of the following phases:

- **Background Review and Related Work:**
  This phase sets a comprehensive background of the involved research topics and reviews the literature of the field of my research. The justification of my choice of the technique I adopt to conduct this research is made. The discussion of the related work addresses the gap and shows how our approach is going to serve the purpose of this study. My motivation towards the selection of the case study is presented at the end of this phase and linked to the related work.
CHAPTER 1. INTRODUCTION

• **Computational Model:**
  As the proposed approach is formal-method based, this phase provides an abstract formalisation for the proposed model. I use Interval Temporal Logic (ITL) and its executable subset Tempura as a formal framework for specification and verification to model the behaviour of parallel systems using the runtime verifier, AnaTempura.

• **Architecture Framework:**
  In this phase I describe the design and the components of the framework. I define each component and describe their functions to be able to link the whole framework and make it integrated. I also introduce different concurrency mechanisms, communication models, and execution modes of parallel computing systems. In addition, this phase presents algorithmic descriptions of all possible approaches that might have been encountered during my design of the proposed framework.

• **Implementation:**
  This phase implements the proposed framework using the programming temporal language Tempura to model the framework components and their functions. Assertion points mechanism is also implemented and a set of practical improvements has been made to enable our framework to handle receiving multiple assertion data sent from local programs simultaneously to compose a global property out of local properties. Java Remote Method Invocation (RMI) framework is implemented as a demonstration of the robustness of my proposed framework and its capability to implement various parallel systems architectures.

• **Case Study:**
  As my approach is formal-methods based, this phase provides an abstract formalisation of the case study which is cache controller. I use Interval Temporal Logic (ITL) as the formal framework for the specification and verification to model the behaviour of cache controller. The cache controller modelled using Tempura and the runtime verifier AnaTempura. A set
CHAPTER 1. INTRODUCTION

of formal ITL specifications transformed into an executable code in Tempura to be checked against a set of temporal properties.

• **Evaluation:**
  The evaluation illustrates the effectiveness of the framework by producing a runtime verification of the cache controller. The correctness is the success measurement of my approach. Correctness of the framework shows consistency of the implemented system.

1.5 Success Criteria

The success criteria of my approach, in addition to answering the research questions, are reflected in the ability of achieving the following outcomes:

1. Compositional requirements from several sources which handle local and global systems correctness. The fulfilment of this success criteria is the answer of the question number 1.

2. Compositional collection of assertion data from several sources to handle True/Interleaving Concurrency associated with Shared-Variable approach. The fulfilment of this success criteria is the answer of question number 3.

3. Compositional collection of assertion data from several sources to handle Synchronous/Asynchronous Communication links, which are *Channels/Shunts*, associated with Message-Passing approach. The fulfilment of this success criteria is the answer of question number 3.

4. The ability to execute agents concurrently and the introduction of resource allocation agents, and Delay and Timeout agents to model delay and timeout behaviour. The fulfilment of this success criteria is the answer of question number 3.

5. The use of lock-based technique to enforce Mutual Exclusion to deliver synchronisation. The fulfilment of this success criteria is the answer of questions number 1 & 3.
CHAPTER 1. INTRODUCTION

6. Checking the correctness property of local systems at local/global levels. The fulfilment of this success criteria is the answer of questions number 2 & 4.

7. The inference of the correctness of global property from the correctness of a set of local properties of global systems. The fulfilment of this success criteria is the answer of the raised major question.

1.6 Thesis Outline

This thesis is composed of five chapters which are organized as follows:

- **Chapter 2** introduces the basic concepts of the relevant topics of verification techniques, runtime verification, the monitor, syntax and semantics of Interval Temporal Logic (ITL) and its executable subset Tempura as a temporal programming language used to model my framework. Also, the justification of my choice of Interval Temporal Logic (ITL) is to serve as a formal-method base of the proposed approach.

- **Chapter 3** introduces the proposed framework and its components and a comprehensive description of the components and their functions. Also, it introduces different concurrency mechanisms, communication models, and execution modes of parallel computing systems. This chapter presents algorithmic descriptions of all possible approaches that were encountered during the design of the proposed framework.

- **Chapter 4** introduces the implementation of the proposed framework and explains in details how to implement the framework components and their communication with each other. Also this chapter implements other parallel computing architectures.

- **Chapter 5** provides the evaluation of the implemented proposed framework. Cache Controller case study is modelled and implemented to evaluate my framework concerning correctness property of cache controller. In addition to the raw data analysis, a formal
specification in Interval Temporal Logic (ITL) of all the components of cache controller system is given. Demonstration of the implemented case study is given in screen-shots as the monitoring system of AnaTempura has an animation window to simulate the system behaviour visually.

- **Chapter 6** provides random and independent evaluation techniques using MATLAB. AnaTempura has been linked to MATLAB in order to exchange assertion data. These assertion data can be used within MATLAB for manipulation, analysis and making unbiased judgement of the proposed model.

- **Chapter 7** summarises the proposed approach and highlights the significance of the delivered contributions and draws a comparison with related work. It also discusses the limitations of my approach, the directions of the future work, and the impact on academic and industrial perspectives.
Chapter 2

Verification Techniques for Parallel Systems: a Review

Objectives:

• To highlight basic Concepts and Related Topics

• To give an overview of Runtime Verification

• To discuss major challenges in Runtime Verification for Parallel Programs

• To investigate relevant Formal Approaches

• To highlight the Related Work, Memory Models, and recent challenges in the field
2.1 Introduction

In this chapter, I present a comprehensive background of the research topic. I present verification techniques which are used in this field and a trade-off between these techniques. I focus mainly on runtime verification to serve the proposed approach due to specific reasons. Interval Temporal Logic (ITL) serves as a formal-method based framework for my approach due to various reasons. I shed light on related work concerning memory models in addition to hardware vulnerabilities, namely, Meltdown and Spectre.

2.2 Basic Concepts and Related Topics

In this section, a brief review of some essential technical aspects are covered such as the difference between concurrency and parallelism, parallel and concurrent models in Java programming language, modern Central Processing Units (CPUs) and Petri Net.

2.2.1 Concurrency versus Parallelism

As in this research parallel systems are intended to be studied in order to deliver correctness properties, I have to clarify the confusion between the terminology of concurrency and parallelism. These terminologies are often debated among computer science communities. The ambiguity in the difference between them is confusing which might lead to misconception in views. Navarro et al. [205] realised this misunderstanding between the two terminologies; hence, they give the following definitions:

**Definition 1**  
“Concurrency is a property of a program (at design level) where two or more tasks can be in progress simultaneously.”

**Definition 2**  
“Parallelism is a runtime property where two or more tasks are being executed simultaneously.”

According to Navarro et al. [205] it is totally different being in progress (concurrency) from being in execution (parallelism). Let $C$ and $P$ denote concurrency and parallelism respectively,
the relationship between them can be expressed formally as: $P \subset C$, which means $P$ is a subset of $C$ and subsequently $C$ is a superset of $P$. In simple words, parallelism is concurrency’s dependent while concurrency is independent of parallelism. Now the difference is even more clearer which invites the definition of a very essential terminology in this research, which is parallel computing:

**Definition 3** “Parallel Computing is the act of solving a problem of size $n$ by dividing its domain into $k \geq 2$ (with $k \in N$) parts and solving them with $p$ physical processors, simultaneously.”

where $k$ represents the least number of processors which is 2. The problem of size $n$ is used to divide the tasks on the available processors in order to achieve the parallel computing consistently and quickly.

### 2.2.2 Parallel-Concurrent Programming Models in Java

Java programming language has considered concurrency since the release of Java 5 by adding the concurrent utilities or alternatively referred to as the concurrent API, where API stands for Application Programming Interface. The concurrency utilities provide powerful features in order to achieve concurrent programs, features and mechanisms such as semaphore, cyclic barriers, countdown latches, thread pools, execution managers and locks.

Java continues to support concurrent programming models such as the introduction of Fork/Join framework to Java 7 release. Fork/Join framework is an implementation of the `ExecutorService` interface that helps to take advantage of multiple processors. The mechanism followed in this Fork/Join framework breaks down the task into smaller pieces recursively and then reassembles them once the task is done. This mechanism enhances the performance of the application via using all available processors.

According to Schildt [234], there are two ways in which Fork/Join enhances multithreaded programming. The first one is the creation of multiple threads, which makes it simple, and the second one is the use of multiple processors, which makes it automatic. However, subdividing or
partitioning the problems with Fork/Join framework must be done by the programmer. The application of operations aggregation by Java runtime performs this task instead of the programmer and puts together the solutions.

The use of the collections mechanism leads to a situation called non-thread-safe which makes the implementation of parallelism difficult. Consequently, thread interference or memory consistency errors are encountered as a result of the incapability of threads manipulating a collection. In order to overcome these errors, synchronisation wrappers provided by the Collection Framework adds automatic synchronisation to collection and makes it thread-safe. Nevertheless, synchronisation wrappers cause thread contention which does affect the parallel execution. In order to implement parallelism with non-thread-safe collections, aggregate operations and parallel streams are used. Executing streams in parallel has been introduced in Java 8. This mechanism allows streams to be executed in serial or parallel. In case streams are executed in parallel, the streams are partitioned by the Java runtime. Aggregate operations iterate over and the results are combined after processing these substreams in parallel.

The latest Java version is Java 12 which was released on 19 March 2019. Java 12 provides concurrency and parallelism utilities such as `java.util.concurrent`, `java.util.concurrent.atomic` and `java.util.concurrent.locks`. The first utility provides concurrent programming. The second utility is a small toolkit of classes that supports lock-free and thread-safe programming on shared memory model. Atomic package provides atomic region which prevents interference of a shared memory being executed within this region in order to guarantee shared memory consistency. The third utility is a set of interfaces and classes that provides a framework for the application of locks mechanism. Locks are responsible for keeping the shared variable protected from multiple modifications at a single clock in order to provide consistent parallel computation. No other processors or threads are permitted to modify the locked shared variable until the lock is released.
2.2.3 Modern Central Processing Units (CPUs)

According to Oak Ridge National Laboratory at the U.S. Department of Energy, June 8, 2018, Summit is the world’s most powerful and smartest scientific supercomputer. Summit can perform up to 200,000 trillion calculations per second or alternatively 200 petaflops. Up until June 2019, Summit supercomputer kept being the first on the list of supercomputers ever since its release with a total number of 2,414,592 processors [1]. IBM developed Summit or OLCF-4 supercomputer for use at Oak Ridge National Laboratory for scientific research.

National Supercomputing Center in Wuxi, China has developed a supercomputer called Sunway TaihuLight and it is ranked third on the list of supercomputers although it has 10,649,600 processors [1]. More numbers of processors does not always mean better performance. Therefore, Sunway TaihuLight supercomputer has the maximum number of processors in the 500 list available in [1].

While the maximum number of parallel processors for modern CPUs at the personal usage level such as PCs or Laptops is 18 processors as Table 2.1 illustrates:

<table>
<thead>
<tr>
<th></th>
<th>No. Processors</th>
<th>Release Date</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i9-9980XE</td>
<td>18</td>
<td>Q4’18</td>
<td>9th</td>
</tr>
<tr>
<td>AMD Ryzen 7 3800X</td>
<td>8</td>
<td>Q3’19</td>
<td>7th</td>
</tr>
</tbody>
</table>

Intel Core i9-9980XE processor is a 9th generation processor and has been released in the forth quarter of 2018 [4]. Ryzen [3] is a 7th generation processor manufactured by Advanced Micro Devices (AMD). The maximum number of processors of AMD Ryzen 7 3800X is 8. Ryzen 7 3800X has been released in the third quarter of 2019.
2.2.4 Petri Net

A Petri Net is a graphical and mathematical modelling tool used to describe and study information processing systems of various types [200, 223, 102]. In 1962, Carl Adam Petri presented this approach as a PhD dissertation entitled "Communication with Automata" in the faculty of Mathematics and Physics at the Technical University of Darmstadt in West Germany. The tool can be used in mathematical branches such as algebraic equations and state equations. Moreover, computer science and communication systems such as logical systems can be modelled and analysed using Petri Nets. Parallel computing has been significantly advanced by Petri’s work. Additionally, modern studies of complex systems have been boosted by Petri Nets approach.

There are a number of different scenarios and applications where Petri Nets are particularly useful in modelling such as state machine, formal languages, multiprocessor systems, dataflow computation, communication protocols, synchronisation control and producers-consumers system with priority [200]. For instance, parallelism can be modelled using Petri Nets as Figure 2.1 illustrates.

![Figure 2.1: Modelling Parallelism using Petri Net [200]](image)
The model considers a parallel system that does a certain computation and at some point the parallel system divides the computation into two execution paths. Each execution path is assigned to a single thread and works independently. Once these two executions are done, at some point they get combined together as one system as they were before the split. This parallel behaviour model helps to enhance the process of designing algorithms and analysis for parallel computing systems.

2.2.5 Global State Construction

Parallel and distributed systems concern the consistency of their composed global states out of local states information as they have multiple subsystems running in parallel. This issue has been considered in [32, 33, 34, 35, 36, 37, 167, 274]. Automatic collections of the information is produced from parallel systems in order to construct global state of the whole system.

Borkowski et al. [34, 35, 36, 37] proposed a methodology of organising execution control in parallel and distributed systems which monitor system global states. Automatic collection of information from parallel systems about their local states and subsequently global states are constructed and composed. The global state can then be evaluated and measured in order to fulfil a desired correctness property of parallel and distributed systems. Borkowski [32, 33] sheds light on the importance and effectiveness of parallel and distributed systems by underlying Consistent Global State (CGS) and Strongly Consistent Global State (SCGS) monitoring mechanisms.

Mattern et al. [167] argue that a structure of linear order of time is not always adequate for a distributed system. Subsequently, a generalised non-standard model of time which consists of vectors is proposed. Timestamps and simple clock update mechanism are used to represent a global time consistently. In order to compute a consistent global snapshot of distributed systems, a new algorithm is proposed in this in their work.

Tudruj et al. [274] propose a control infrastructure which is based on synchronisers organised at the processors and threads levels in order to collect local states information for evaluation sake
and produce a consistency model of global state for parallel and distributed systems.

The construction of global state of a system out of local states of subsystems which run in parallel or in distributed model is significant and it enhances the verification process of such systems. Correctness properties of subsystems can be fulfilled and generalised using construction mechanism of global states. The global state’s correctness property of the parallel/distributed systems $Prop$ can be constructed from the set of the correctness properties of their subsystems $Prop_i$. This assertion can be expressed formally as follows:

\[
\bigwedge_{i=0}^{i=n} Prop_i \supset Prop
\]

where $i$ represents the identification number $id$ of processors or threads assigned to execute subsystems run in parallel or in distributed model. The local correctness property of processor or thread $i = 0$ is $Prop_0$ and $i = n$ is $Prop_n$, while the global correctness property is $Prop$.

It can been seen theoretically so far that a construction of global state out of local states for parallel and distributed systems is possible. Therefore, local states correctness properties can infer the correctness property of global state. A practical demonstration of this assertion will be presented later in this thesis.

2.2.6 Parallel Programming Models

The difference between parallel computing models and parallel programming models is that the former concerns designing parallel algorithms and analysing technical aspects, for instance, computing time complexity. Some commonly used parallel computing models for such purposes are PRAM [94], (U)PMH [9], BSP [277] and LogP [67]. These prior parallel computing models will be discussed later in this research. While parallel programming models concern the communication aspects of parallel processors and how they should be programmed. The most important parallel programming models are Shared Memory and Message Passing due to their wide use
and implementation by modern Application Programming Interfaces (APIs).

2.2.6.1 Shared Memory

Reading and writing to a shared memory using this programming model is asynchronous. Shared memory model is natively useful for multicore systems. Non-deterministic behaviour of parallel processors have to be managed using parallel algorithms for this sake in order to maintain shared memory consistency model. Read and write operations on the same shared memory are possible at any time, and an explicit synchronisation and control mechanisms have to be applied such as monitors, semaphores, atomic operations and mutual exclusion (*mutex*). These synchronisation solutions and shared resources control mechanism enable processors to lock a shared memory in order to get a consistent copy. Once the shared memory is locked, no other processors are allowed to interfere. Constraints on shared memory can be applied in order to guarantee shared memory consistency model [89]:

- All processors/threads must see exactly the same values for a shared memory;
- All processors/threads must see updates to the memory at the same time;
- Only one processor/thread is allowed to write to the shared memory at any given time.

2.2.6.2 Message Passing

Message passing programming model allows processors to communicate asynchronously or synchronously. Processors can send and receive messages containing words of data. Messages might arrive *i*) very quickly, *ii*) within a fixed period of time, *iii*) at some point of time in the future, or *iv*) possibly never in case errors are encountered. There are various forms where processors can communicate; the most common three mechanisms are:

- Point-to-Point, where the communication occurs only between two processors, the sender and the receiver;
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

- Broadcast, where the sender sends the message without a certain destination;

- Multicast, where messages can get delivered to subsystems with some restrictions.

Message Passing Interface (MPI) is the standard interface for message passing model. MPI can be used to distribute the work and handle communication in CPU distributed applications.

2.2.6.3 Shared Memory versus Message Passing

It can be seen that the implementation of shared memory is practically less complicated, less time consuming and can be done automatically. On the other hand, the implementation of message passing model is more complicated, time consuming, and has to be done manually by the programmer. Parallel programming models community believes in the fact that shared memory model has superiority over message passing model. Therefore, I will omit the implementation of message passing model in my proposed model for communication aspects. Alternatively, shared memory model will be implemented only.

2.3 Verification Techniques

Verification is a process of checking whether a system under scrutiny is acting accordingly to the contract which has been signed between that system and a set of desired properties to guarantee the correctness criteria. Mainly, the verification process has different techniques such as theorem proving [30], model checking [63], testing [40, 202], and runtime verification [150, 151] as Figure 2.2 illustrates.

Theorem proving is a correctness of programs using mathematics proof to deliver correctness of a theorem, and it is primarily applied manually. Model checking is an automatic verification, and it checks all the possible states of finite systems which lead to states explosion. Testing can be classified as incomplete verification techniques for checking correctness. However, runtime verification complements between some of these verification techniques such as model checking and testing. Runtime verification monitors system behaviour only at runtime which avoids
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

These verification techniques are subject to some factors such as the availability of formal model for model checking and confidence strength or weakness in favour for theorem proving over testing. In comparison, runtime verification technique is considered a lightweight verification technique. Verification techniques such as model checking and testing can get complemented by runtime verification. Due to the nature of such verification technique, it occurs at runtime and only explores states which are being executed. This characteristic merits runtime verification over model checking and testing throughout the absence of states explosion of model checking and incompleteness of testing technique.

![Figure 2.2: Verification Techniques](image)

2.4 Runtime Verification

Static verification techniques assume that designed models are completely explorable in order to deliver correctness of these models. The assumption of the ability to access and explore such models is not reasonable. Therefore, verification techniques must offer runtime verification techniques. The states of systems under scrutiny are then generated and collected in order to do the
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

analysis of their behaviour and make the judgement accordingly. Runtime verification technique complements static verification techniques [177]. The term correctness is defined as follows [29]:

**Definition 4** “In computing systems, correctness refers to the assertion that a system satisfies its specification.”

Runtime verification checks whether the execution of a system complies with the correctness criteria involved within the process of system design to meet a set of desired properties such as safety, liveness, and projected time.

Runtime verification has the ability to handle inadequacy of information being executed because it is intended to observe the executed information only and provides a property check against the correctness specification that is already prescribed formally and internally within the runtime verifier. A runtime verifier should not infer the execution of what is being executed so far; in other words, it should not enforce an execution of a certain instance of a system under scrutiny if it is not yet reached. Alternatively, it only detects the violation of the correctness criteria and checks whether the prescription of the system is being respected especially for on-the-fly applications [177].

When a variation between the required behaviour and the observed behaviour of the system under scrutiny occurs, it is called a system failure. In case the expected behaviour and the current behaviour of a system under scrutiny are not matched, it is called fault. When a human makes a mistake, it is called an error. Potentially, a failure can be caused due to a fault; similarly, a fault can be caused due to a mistake.

According to IEEE [2], verification concerns the techniques which are intended to show a system satisfaction with its specification. Verification techniques such as theorem proving [30], model checking [63], and testing [40, 202] are considered traditional verification techniques. Runtime verification is introduced as a new direction within the field of verification techniques.
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

**Definition 5** “Runtime Verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [151].”

Runtime verification [65, 216, 103] uses a device called monitor which checks at runtime the satisfaction or the violation of the execution of system under scrutiny against a correctness temporal property. The traditional verification techniques such as theorem proving, model checking, and testing are complemented by runtime verification.

Runtime verification does not influence or interfere with the execution of a system under scrutiny in case a correctness property is violated; instead, it only deals with the detection of either violation or satisfaction of correctness property.

### 2.4.1 Monitors

The only concern for runtime verification is whether the run of a system $Sys$ is satisfied against a correctness property $\varphi$. A monitor is intended to check whether the execution of a system $Sys$ satisfies a correctness property $\varphi$. When a correctness property $\varphi$ is met, a truth value is dispatched. Formally, a set of valid executions $[\varphi]$ is given by property $\varphi$, and runtime verification checks whether the execution of $Sys$ belongs to a set of valid executions $[\varphi]$ [151].

**Definition 6** “Monitor is a device that reads a finite trace and yields a certain verdict [151].”

A verdict being yield is a truth value which belongs to a truth domain $\mathbb{B} = \{true, false\}$ or alternatively it might have this form $\mathbb{B} = [0, 1]$. The truth value $true$ or $1$ denotes that a correctness property $\varphi$ is satisfied; otherwise, a correctness property $\varphi$ is violated [151].

### 2.4.2 Taxonomy

Runtime verification has brought different contributions into the verification techniques field. Aspects of runtime verification are systematically presented as Figure 2.3 illustrates. The aspects are Trace, Monitoring, Stage, Integration, Interference, Steering, and Application Area [151].
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

Figure 2.3: Taxonomy of Runtime Verification [151]
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

**Trace:** Runtime verification has the ability to work on *i*) finite (terminated), *ii*) finite but continuously expanding, or *iii*) on prefixes of infinite traces. In regards to finite but continuously expanding and infinite traces, impartiality and anticipation should be taken into the monitor’s account. Impartiality implies that judgement should not be made of a finite trace in case there is an infinite continuation trace which might lead to a different verdict. Anticipation implies that a judgement should be made if an infinite continuation of a finite trace has the same verdict value.

**Monitoring:** Runtime verification has different interests or concerns in terms of what is being monitored. For instance, a system may get checked in terms of the input or output behaviour. Moreover, a system may get checked in terms of sequence of states, or in terms of sequence of events being executed.

**Stage:** Online monitoring occurs when a current execution of a system is being checked by a monitor. Offline monitoring occurs when the execution of a system being checked is recorded.

**Integration:** Inline monitoring occurs when a monitoring code is interwoven with the code of a program to check. If the monitoring code is used to externally check a program under inspection, then the monitoring is outlined.

**Interference:** Invasive monitoring interferes with the system being checked, while non-invasive monitoring does not interfere with the system being checked.

**Steering:** When a monitor only observes the program execution and reports program failures, then it is called passive monitoring. When the monitor is used to steer the program execution, it is called active monitoring.

**Application Area:** Runtime verification serves different application and purposes. It might be used to check safety or security conditions. It can also be used to collect information of the system being executed for performance evaluation purposes.
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

2.4.3 Runtime Verification versus Model Checking

Model checking determines whether all computations of a given model $M$ satisfies a correctness property $\varphi$. Model checking can be considered an automatic verification technique which can handle finite state systems. In automata theoretic approach [279], a transformation of correctness property $\varphi$ to an automaton $M_{\neg \varphi}$ which accepts a violation of a correctness property $\varphi$. The automaton $M_{\neg \varphi}$ is run in parallel to a model $M$ in order to check whether $M$ is violating a correctness property $\varphi$. Similarly, runtime verification has to generate the monitor as the model checking has to generate an automaton. However, there are differences between them:

- Model checking examines all possible executions of a given model of a system $M$ whether the executions of the model $M$ satisfies a correctness property $\varphi$. However, runtime checks only the execution of a model $M$ at runtime whether this runtime execution satisfies a correctness property $\varphi$.

- Model checking considers infinite traces, while in runtime verification only finite executions are considered.

- As a consequence of considering infinite traces by model checking, the state explosion problem is encountered. On the other hand, a single run of a system does not cause this problem.

2.4.4 Runtime Verification versus Testing

As stated above, runtime verification does not check the whole possible execution of a system under scrutiny; instead, it just checks the single execution of a system. This characteristic makes runtime verification and testing both incomplete verification techniques.

Testing receives finite input-output sequences which form what is called test suite [217]. Then the checking process takes place whether the actual output and the expected ones are met or not. Another form of testing which is relatively closer to runtime verification is called oracle-
based testing. The difference between this test and the former one is that the test suite is only formed by input sequences. Then an oracle-based test has to be designed and attached to the system being tested to make sure that the output is anticipated. Runtime verification can be seen from this angle; however, there are differences between these verification techniques:

- In testing, more precisely oracle-based test, an oracle is defined directly rather than getting it from a generation of high-level specification.

- In testing, in order to test a system exhaustively, input sequences have to be provided. In contrary, this is considered internally within a domain of runtime verification.

Therefore, runtime verification can be considered a passive form of testing in addition to the fact that runtime verification tests forever which makes it complete.

2.4.5 The Use of Runtime Verification

The model checking and theorem proving reflect the most important aspects of the implementation via a model check and analysis to make sure the implementation meets the correctness property. However, the implementation, due to the environment surrounding the system under scrutiny, might behave differently from what is being predicted by the model. Runtime verification is then used to overcome this obstacle via a runtime check of the actual execution of a system under scrutiny and find out whether a correctness property is satisfied. Therefore, runtime verification in this case can be considered as a partner to model checking and theorem proving [151].

There are cases where some information of a system under scrutiny can only be available at runtime and can not be explored using other verification techniques. The information of a system is not only available at runtime. However, this information is checked out at runtime because it is more convenient than checking it using different verification techniques, due to the nature of the system under scrutiny [151].
The environment influences the behaviour of a system being executing; therefore, the environment of a system under scrutiny matters. Model checking and theorem proving make assumptions on the behaviour of a system within a certain environment. However, the assumptions made by model checking or theorem proving is inadequate. Therefore, runtime verification performs a formal correctness of these assumptions [151].

For critical systems security and safety aspects, it might be beneficial to monitor a system which has already been checked to make sure it adheres to the correctness property that is already met. In this case, runtime verification can be considered as a partner to model checking, theorem proving, and testing [151].

Due to the critical role of runtime verification and its partnership with other verification techniques, runtime verification is worthy to be considered a major verification techniques and be a fundamental component of the architecture of system designs.

2.4.6 Existing Runtime Verification Frameworks

Martin Leucker visited in [150] the considered existing runtime verification frameworks. Some of these frameworks are considered major players in the field such as EAGLE, J-LO, Larva, LogScope and LoLa.

2.4.6.1 EAGLE

EAGLE [23] is a rule-based framework intended to define and implement finite trace monitoring logics, such as future and past time temporal logic, extended regular expressions, real-time logics, interval logics, and forms of quantified temporal logics. EAGLE’s novel techniques for rule definition, manipulation and execution are implemented as a Java library. Monitoring mechanism follows a state-by-state basis, without storing a trace of the execution.

2.4.6.2 J-LO

J-LO [263] is a runtime verification framework for Java programs. The specification of properties can be formally expressed in Linear-time Temporal Logic (LTL) over AspectJ pointcuts. The
automaton-based approach where transitions can be triggered via aspects is used to check these properties expressed in LTL at runtime. As AspectJ is working on the bytecode level, Java source code is unnecessary.

2.4.6.3 LARVA

LARVA [66] is a runtime verification framework and is considered a lightweight approach to guarantee properties of Java programs including real-time properties. LARVA enables properties to be expressed in formal notations such as timed-automated, Lustre and a subset of the duration calculus. The tool has been used as a case study for industrial systems, and it has been successfully working. At the analysis level of real-time properties, LARVA performs as well as calculates memory and temporal overheads caused by monitoring process. The tool is also used in order to assess the consequences caused by the process of monitoring such as slowing down a system in order to satisfy the desired properties of a system.

2.4.6.4 LogScope

LogScope [25] is a Python framework that allows to check logs for conformance to a specification and to learn patterns from logs. LogScope architecture divides its functionality into LogMaker tool and a core LogScope module. The latter checks logs and learns specifications. LogScope is developed by and dedicated to NASA’s Mars Science Laboratory Mission (MSL). A list of events is generated by LogMaker and after a communication channel is opened to MSL’s SQL-based ground software. LogScope receives two arguments i) a log generated by LogMaker, and ii) a specification. The specification language offers an expressive rule-based language, which supports state machine, a higher-level pattern language, which is then translated into a more expressive rule-based language in order to perform the monitoring process. Logging systems events can be used as a basis for automated evaluation of log files against requirements.
2.4.6.5 LoLa

LoLa [69] is a specification language and algorithms for online and offline monitoring of synchronous systems such as circuits and embedded systems. Despite the simplicity of the specification language, it is elegantly expressive. It can be used to not only describe correctness property but also detect failure by using assertions, so a measurement of interesting statistics can be used for system profiling and coverage analysis. The language has been used for monitoring industrial systems such as Peripheral Component Interconnect (PCI) bus protocol and memory controller. The outcomes prove that the specification language is sufficiently expressive in such systems and applications.

2.5 Formal Methods-Based Tools for Parallel Systems

Verification techniques for parallel systems require formal-methods based tools which use mathematical concepts such as formal semantics, formal specification, and formal verification to check the desired correctness property of such systems. The most common correctness properties of the execution of parallel systems is concurrency errors such as data races, deadlocks, livelocks, atomicity violation. Formal-methods based techniques are applied such as deductive verification (theorem proving), model checking, static program analysis, and runtime verification. As I discussed in the previous sections, 2.4.3, 2.4.4, & 2.4.6, that runtime verification complements other verification techniques such as theorem proving, model checking, and testing; thus, my interest in this research is a runtime verification due to the discussion above and the reasons listed in the previous sections.

The evaluation of my approach, runtime verification of parallel systems, focuses on the following criteria: [199]

- **Soundness**: A verification technique is considered sound when the check results are valid with respect to the semantics of the programming language, or simply when none of the errors of an execution of a system is missed.
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

- **Completeness**: A verification technique is considered complete when it omits the production of false positives because each detected error requires an investigation which implies a human intervention.

- **Modularity**: A verification technique is considered modular when it has the ability to deduce the correctness of the whole system from the correctness of its independent components. Modularity allows to analyse and check parallel systems.

- **Automation**: A verification technique is considered automatic when it requires no human intervention. A verification technique might be considered highly automatic if it requires little human intervention. Human intervention includes, for instance, providing system specification to be checked.

- **Efficiency**: A verification technique is considered efficient when it has the ability to check large systems in short amount of time and space.

### 2.6 Temporal Logic

According to Konur [139], Temporal logics are formal frameworks which describe statements whose truth values change over time. In comparison with classical logics, temporal logics characterise the change of states over time where classical logics do not include time constraints. The introduction of time characteristics in temporal logics makes it richer than classical logics.

Temporal logics have been widely used for more than two decades in the field of various systems specifications, such as real-time systems and control systems (sequential or parallel manners). Temporal logics use mathematical notation in order to deliver formal analysis and model for systems. Temporal logics have been applied in industrial application and academic disciplines [139].

Temporal logics are introduced in order to solve specific problems that cannot be completely solved using different logics either due to the expressiveness or complexity issues. Expressive-
ness and complexity are the main trade-off concerning temporal logics. The use of temporal logics is subjected to these trade-offs; some applications prefer expressiveness over complexity, while other applications prefer the complex over expressiveness [28].

The classification of temporal logics can be based on various dimensions such as propositional versus first-order logic, point-based versus interval-based, linear versus branching, and discrete versus continues [28, 83, 281]. In the next section, I discuss why interval-based temporal logics is more expressive than point-based temporal logics. I omit the discussion of other dimensions which temporal logics can be based on due to the fact that I adopt Interval Temporal Logic (ITL) [51] as a formal methods-based framework for my research; therefore, I discuss only this dimension to justify my selection of Interval Temporal Logic (ITL) over other temporal logics.

### 2.6.1 Point-Based versus Interval-Based Structure of Temporal Logics

Modelling time in temporal logics has two structures, either point-based structure or interval-based structure. Different modal operators are used to describe different temporal relationships. Some temporal logics use modal operators to express quantification over time. However, point-based temporal logics tend to be difficult to express relationships between intervals [86].

Point-based temporal logics such as Linear Temporal Logic (LTL) [214], Computational Tree Logic (CTL) [62, 84, 145] are used formulas to specify desired properties. These logics are suitable to model computation states and their relationships. however, they are not suitable to model a computation stretches such as actions with durations, accomplishment, and temporal aggregations. Interval-based logics can overcome these limitations of point-based logics via the consideration of time as intervals not points [178].

Interval Temporal Logics (ITLs) are temporal logics which are intended to reason about periods of time (intervals). The representation formalisms of time as intervals are more expressive than formalisms as points. Interval-based logics enrich representation formalisms more than
point-based logics. This enrichment allows interval-based logics to be used to model real-time systems behaviour [139].

Expressiveness of interval temporal logics enables them to express a relationship between events modelled using intervals. The syntax of interval temporal logics [237, 238, 193, 144, 169, 221, 110] is simpler and neater than point-based logics. The syntax of interval temporal logics enables them to provide high level abstraction in order to model systems. Therefore, interval temporal logics formulas are more comprehensive than point-based logic formulas.

Table 2.2: LTL vs CTL vs ITL

<table>
<thead>
<tr>
<th>Logic</th>
<th>Logic Order</th>
<th>Fund. Entity</th>
<th>Temporal Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL</td>
<td>Propositional</td>
<td>Point</td>
<td>Linear</td>
</tr>
<tr>
<td>CTL</td>
<td>Propositional</td>
<td>Point</td>
<td>Branching</td>
</tr>
<tr>
<td>ITL</td>
<td>First-order</td>
<td>Interval</td>
<td>Linear</td>
</tr>
</tbody>
</table>

Table 2.2 compares point-based logics such as Linear Temporal Logic (LTL) [214] and Computational Tree Logic (CTL) [62, 84, 145] with interval-based logics such as Interval Temporal Logic (ITL) [193]. The main criteria of comparison is the representation of time in either points or intervals form.

2.6.2 Interval Temporal Logic (ITL)

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems [51]. Interval Temporal Logic (ITL) can handle sequential and parallel compositions, and it has a powerful and extensible specification and proof techniques in order to reason about properties such as safety, liveness and projected time [194]. ITL has the ability to express timing constraints and most imperative programming constructs as well can be expressed as formulas in an executable modified version of ITL called Tempura [47]. Tempura is an executable subset of ITL, and it provides an execution framework for ITL specifications to shift a system from abstract specification to
concrete implementation. In addition, ITL and its mature executable subset Tempura [182] have been extensively used to specify and model the properties of real-time systems where the primitive circuits are directly represented by a set of temporal formulae. Tempura has been applied variously to simulate hardware design and other areas where timing is crucially important.

### 2.6.2.1 Syntax

The key notion of ITL is an interval. An interval $\sigma$ is considered to be a (in)finite sequence of states $\sigma_0, \sigma_1, \ldots$, where a state $\sigma_i$ is a mapping from the set of variables $Var$ to the set of values $Val$. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to $n$ (one less than the number of states in the interval (this has always been a convention in ITL), for instance, a one state interval has length zero [51]. The syntax of ITL is defined in Table 2.3, where:

- $z$ is an integer value,
- $a$ is a static integer variable (doesn’t change within an interval),
- $A$ is a state integer variable (can change within an interval),
- $v$ a static or state integer variable,
- $g$ is a integer function symbol,
- $q$ is a static Boolean variable (doesn’t change within an interval),
- $Q$ is a state Boolean variable (can change within an interval),
- $h$ is a predicate symbol.

#### Table 2.3: Syntax of ITL

<table>
<thead>
<tr>
<th>Expressions</th>
<th>( e ::= )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( z \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \oplus A \mid \text{fin} \ A )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Formulae</th>
<th>( f ::= )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{true} \mid q \mid Q \mid h(e_1, \ldots, e_n) \mid \neg f \mid f_1 \land f_2 \mid \forall v \cdot f \mid \text{skip} \mid f_1 : f_2 \mid f^* )</td>
</tr>
</tbody>
</table>
2.6.2.2 Informal Semantics

The informal semantics of the most interesting constructs are as follows: [51]

- $\bigcirc A$: if interval is non-empty then the value of $A$ in the next state of that interval else an arbitrary value.

- $\text{fin } A$: if interval is finite, then the value of $A$ in the last state of that interval else an arbitrary value.

- $\text{skip}$ unit interval (length 1).

\begin{align*}
\text{skip; } X &= 1 \quad \bullet \quad \bullet \quad \bullet \quad \bullet \\
(\bigcirc X &= 1) & \quad X: 2 \quad 1 \quad 2 \quad 4 \\
\text{finite; } X &= 1 \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \\
(\bigdiamond X &= 1) & \quad X: 1 \quad 1 \quad 4 \quad 1 \quad 1 \\
\neg(\text{finite; } X &= 1) \quad \bullet \quad \bullet \quad \bullet \quad \bullet \\
(\Box X &= 1) & \quad X: 1 \quad 1 \quad 1 \quad 1
\end{align*}

Figure 2.4: Some Sample of ITL Formulae [51]

- $f_1: f_2$ holds if the interval can be decomposed (“chopped”) into a prefix and suffix interval, such that $f_1$ holds over the prefix and $f_2$ over the suffix, or if the interval is infinite and $f_1$ holds for that interval.

\begin{figure}[h]
\centering
\begin{tikzpicture}
    \node (start) at (0,0) [circle,fill,inner sep=2pt] {};
    \node (middle) at (1,0) [circle,fill,inner sep=2pt] {};
    \node (end) at (2,0) [circle,fill,inner sep=2pt] {};
    \node (f1) at (0,1) [circle,fill,inner sep=2pt] {};
    \node (f2) at (2,1) [circle,fill,inner sep=2pt] {};
    \draw[thick] (start) to (middle) to (end);
    \draw[thick] (f1) to (f2);
\end{tikzpicture}
\caption{Chop}
\end{figure}
• $f^*$ holds if the interval is decomposable into a finite number of intervals such that for each of them $f$ holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which $f$ holds.

![Chop Star Diagram](image)

Figure 2.6: Chop Star

### 2.6.2.3 Justification for Choosing Interval Temporal Logic (ITL)

The characteristics of Interval Temporal Logic (ITL) induced its choice. These characteristics are presented as follows [299]:

- **ITL** is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems.

- Unlike most temporal logics, ITL can handle both sequential and parallel composition and offer powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and time.

- Tempura, the executable subset of ITL, provides an executable framework for developing, analysing and experimenting with suitable ITL specifications [182].

- Modular and reusable tempura test suites can be developed.

- Several specifications can be compared over a range of test data.

- The use of specialised theorem provers and model checkers can be postponed until after a preliminary runtime consistency check of candidate specifications and proofs.

- In contrast to model checking, execution can be used to check theorems that are not decidable.
• ITL and Tempura both improve through the increased feedback between theory and practice. Particular benefits are:
  – The discovery of further executable assumptions and commitments specifications
  – The development of more and better compositional proof techniques

• Interval Temporal Logic serves as the single unifying logical and computational formalisation at all stages of analysis.

• ITL has a complete axiomatic system [197].

• In addition, Cau and Zedan [47] have provided a refinement calculus for ITL that can “translate” an ITL formula into an executable code.

2.7 Related Work

A review of the literature has led to a drawback of Interval Temporal Logic (ITL) which is the lack of memory model. Therefore, related suggested memory models such as Framing Variable and Transactional Memory (TM) are investigated in this section in order to diagnose the situation and avoid being trapped by such a limitation.

2.7.1 Memory Models for Interval Temporal Logic (ITL)

One of the most critical issues within the field of parallelism and concurrency of real-time systems is the access to the common shared resource (memory). Due to the importance of this issue, I review some of related work approaches which have been done in order to overcome obstacles which might be encountered in this field. More precisely, approaches such as Framing Variables, and Transactional Memory.

2.7.1.1 Framing Variables

Framed variables remain unchanged at a state, or over an interval, when no assignment is encountered at that state, or over the interval. Framing Variables is defined as follows [295]:

37
CHAPTER 2. VERIFICATION TECHNIQUES FOR PARALLEL SYSTEMS: A REVIEW

**Definition 7** “Framing a variable \( x \) means that the variable \( x \) always keeps its old value over an interval if no assignment to \( x \) is encountered.”

In temporal logics, no value inherited from a previous state. Alternatively, if a value is needed to be inherited, a repeated assignment of the value should take place at every state. To inherit a value during an interval, I use a formula for each relevant variable such as \( \text{stable}(x) \). The repeatability of such assignments affects the efficiency of the program and makes it tedious [190, 295]. The application of such a mechanism [196] makes the specification implicit and neat.

The study of framing variables in ITL [190] is initiated by Hale [107]. An investigation of framing variables has also been done by Duan [295]. Projection Temporal Logic (PTL) is an ITL extension with operators for temporal granularities and framing [78, 295]. Subsequently, an executable subset of PTL called Framed Tempura is introduced [295]. Framed Tempura has new operations such as projection operator \( \text{prj} \), synchronous communication \( \text{await} \), and framing operator \( \text{frame} \) [295].

However, there are cases where an explicit statement has to be made upon a variable that does not change. Whenever a memory cell has to be updated, it will be a very costly operation. This is called the framing problem. As a solution to this problem is an increase of the speed of the simulator. Instead of updating \( m \) memory cells \( m \) times, only one statement is needed [49].

### 2.7.1.2 Transactional Memory

There four attributes which define the transaction notion are Atomicity, Consistency, Isolation, Durability or what is known for short as ACID. Transactional Memory (TM) is defined as follows [82]:

**Definition 8** “Transactional Memory (TM) is a promising lock-free technique that enables parts of a program to execute with atomicity and isolation, without regard to other concurrently executing tasks. TM allows programs to read and modify disparate primary memory locations atomically as a single operation.”

38
Atomicity ensures either a commitment of the operations in a transaction completely or abortion of all the operations and leaving no evidence behind [82].

Consistency ensures that the data in the memory is consistent with its corresponding state. Only successful transactions commit their data and permanently store them; otherwise, the old data is restored. Isolation ensures that an execution of a transaction does not affect other concurrent transactions. In other words, the result of these concurrent executions has to be equivalent like they were executed sequentially. Durability ensures storing the modified data of a successful transaction on a durable media such as a disk.

Transactional Memory (TM) is relatively easy to use, and it does not need locks, as it is lock-free which avoids the occurrence of deadlocks scenario. The performance is boosted due to the increase of parallelism level. However, its application is limited and the debugging is difficult to place a breakpoint within the transaction.

El-kustaban [82] [80] has formalised Transactional Memory (TM) in Interval Temporal Logic (ITL) and verified it using Tempura/AnaTempura. There are still aspects such as nested transactions and mechanisms of updating the memory which should be imported to provable abstract TM.

It is challenging to control parallel systems accessing shared resource in order to guarantee correctness property such as consistency of shared resource. In order to avoid having access conflicts, a synchronisation mechanism has to be applied. Techniques have been used to apply synchronisation mechanism such as lock-based, lock-free and wait-free.

Lock-free and wait-free avoid using locks which could cause deadlock. However, they are complex to implement. More precisely, as Transactional Memory (TM) is a lock-free technique, it avoids lock-based problems and offers high-level abstract parallel programming models. However, even though the claim made by Transactional Memory (TM) research community that programming with Transactional Memory (TM) is easier than alternatives such as locks, but evidence is scant [228]. A study was made [228] in which 147 undergraduate students in an op-
erating systems course implemented the same programs using coarse-grain and fine-grain locks, monitors, and Transactional Memory. A survey was made on the students after the assignment and their code was examined to determine the types and frequency of the programming errors for each synchronisation technique. The evaluation shows that students found Transactional Memory (TM) harder to use than coarse-grain locks, but slightly easier to use than fine-grained locks.

More reasons why Transactional Memory (TM) is not sufficient enough are space overhead and latency. Transactional Memory (TM) requires significant amounts of global and per-thread meta-data. Transactional Memory (TM) has high single-thread latency, usually two times compared to lock-based technique [68]. Generally speaking, Mutual Exclusion (mutex) locks limit concurrency but offers single-thread latency, whereas, Transactional Memory (TM) has higher latency but scales well [68].

2.7.2 Meltdown and Spectre

Meltdown [156] and Spectre [138] are hardware vulnerabilities in modern computers leak passwords and sensitive data. Meltdown and Spectre take advantage of modern processors critical vulnerabilities. As a consequent of these hardware vulnerabilities, programs get permissions to steal data that has been processed on the computer. Although reading data of programs from other programs is not permitted, a malicious program takes advantage of Meltdown and Spectre to get hold of sensitive personal information stored in the memory of other running programs. Stolen information might be passwords, personal photos, emails, bank card details, etc. Meltdown and Spectre might hit personal computers, mobile devices, and cloud servers. Hitting cloud providers’ infrastructure might cause a steal of data from other customers.

Meltdown breaks the most fundamental isolation between user applications and the operating system. Consequently, programs are allowed to access the memory of other programs and the operating system. Spectre breaks the isolation between different applications. Consequently, error-free programs get tricked by an attacker to leak their secrets. Spectre is harder to exploit
than Meltdown, but it is also harder to mitigate. For more information about Meltdown and Spectre, I refer the reader to [156, 138].

These hardware vulnerabilities, Meltdown and Spectre, are my motive of choosing a case study of cache controller of cache memory and its implication on the main memory with respect to their correctness. The case study demonstrates a correctness of such critical systems and parallel architectures such as multicore architecture to deliver modular, sound, complete, automatic, and an efficient model of the proposed computational model.

2.8 Summary

In this chapter a comprehensive background about the research topic is given. Verification techniques is presented and a trade-offs between these techniques is discussed. Runtime verification has been chosen to serve the proposed approach due to specific reasons which have been presented as well. Interval Temporal Logic (ITL) has been chosen to be a formal methods-based framework for the approach due to various reasons. Related works concerning memory models are discussed in addition to hardware vulnerabilities, namely, Meltdown and Spectre.
Chapter 3

Computational Model

Objectives:

- To introduce the Parallel Runtime Verification Framework (PRVF) Model
- To highlight the Communication Mechanisms, Concurrency Forms, and Execution Modes
- To produce Novel Algorithms and establish a Theoretical Ground
- To describe the Components of the Model
- To demonstrate the Capabilities of the Model
CHAPTER 3. COMPUTATIONAL MODEL

3.1 Introduction
In this chapter, the computational model, namely, Parallel Runtime Verification Framework (PRVF) is introduced. A comprehensive description of the main components of PRVF and their functions is given. The framework has two levels which are global level and locals level; and it has three phases which are Generation Phase, Locals Verification & Assertion Phase, and Global Verification Phase. I describe the possible communication models and concurrency forms that the proposed framework is intended to handle. Then a description of these levels and phases of the proposed framework is given. Novel algorithms are invented, described, validated, and implemented in order to establish a theoretical ground for Parallel Runtime Verification Framework (PRVF) model.

3.2 Computational Model
Parallel Runtime Verification Framework (PRVF) is a generic model which is intended to handle several parallel computing characteristics such as concurrency forms, communication models, and execution modes. Concurrency forms might be true or interleaving. Communication models might use shared-variable or message-passing based approach. Execution modes might follow either synchronous or asynchronous mechanism. Therefore, I introduce a framework that can handle both concurrency forms, true concurrency and interleaving concurrency, associated with either shared-variable or message-passing based approach for (a)synchronous communication links. Synchronous communication links are called Channels, while asynchronous communication links are called Shunts [47]. Later in this chapter, a comprehensive description of Channels and Shunts constructs is given.

3.2.1 Message-Passing based Communication
Message-passing based is a model of communication between parallel systems via sending and receiving to/from other systems. Predicates such as send and receive are used to perform com-
munication between systems. A message being sent may either arrive or never arrive due to system failure. In case a system fails, there shall be a timeout option to avoid waiting forever. There are forms of message-passing communication in terms of the sender and the receiver such as the following:

- **Point-to-Point**: It is where a message is sent from one sender to one receiver.
- **Broadcast**: It is where a sender dispatches the message without knowing information about the receivers, such as names and addresses of the recipients.
- **Multicast**: It is where a sender is allowed to broadcast not only to receivers, but also to subset of all possible receivers without knowing the names or addresses of the receivers and their subsets.

3.2.1.1 Related Work

Cau and Zedan [47] extended Interval Temporal Logic (ITL) to include modularity, resources, and explicit communications. This extension [47] allows synchronous, asynchronous and shared-variable concurrency to be explicitly expressed. The developed model in [47] uses the shared-variable approach to model message-passing in Interval Temporal Logic (ITL). The constructs Channels and Shunts are modelled. Channels are synchronous communication links, while Shunts are asynchronous communication links.

The proposed computational model in [47] is closely related to a wide-spectrum language called Temporal Agent Model (TAM) [236]. Temporal Agent Model (TAM) can express both functional and timing properties in either abstract or concrete levels. However, Cau and Zedan [47] introduced timed-communication, timeout, and resource allocation constructs in Interval Temporal Logic (ITL) semantics because the original TAM semantics is not accessible enough. In addition to Channels and Shunts constructs, timing constraints, such as delay and timeout, resource allocation, and shunts’ multiplexer called Funnel are modelled in Cau and Zedan computational model [47].
3.2.1.2 Execution Modes

To discover *Channels* and *Shunts* constructs, modes of execution, such as (a)synchronous, have to be illustrated first. *Synchronous* execution enforces parallel systems to start and finish their execution at the same clock as Figure 3.1 illustrates, while *asynchronous* execution allows systems run in parallel to start and finish their execution at different clocks as Figure 3.2 illustrates where $Sys_1$ and $Sys_2$ represent any system running in parallel; $\sigma_n$ represents the state number.

![Figure 3.1: Synchronous Execution of Parallel Systems $Sys_1$ and $Sys_2$](image1)

![Figure 3.2: Asynchronous Execution of Parallel Systems $Sys_1$ and $Sys_2$](image2)

3.2.1.3 Channel Communication

The variables $C \in Chan$ are the representation of channels whose values are triples $(wtr, wtw, v)$ where:

- $wtr$ is a boolean value and its value indicates whether the system is willing to accept (read) a message from that channel.
**•** *wtw* is a boolean value and its value indicates whether the system is willing to send (write) a message to that channel.

**•** When *wtr* and *wtw* are both true, *v* stands for the value currently in channel *C*.

To introduce a channel *C*, **channel** *C* ∈ *P* is used. To send a value of expression *e* over the channel *C* and denotes an output which has been sent we use *C!* *e*. To receive a value over the channel *C* and store it in *x* and denotes it as an input we use *C*? *x*:

```
channel C in P ≡ ∃ C ∈ P

C? ≡ Π₁(C) = true
C! ≡ Π₂(C) = true
C.x ≡ Π₃(C) = x ∧ C? ∧ C!

C!e ≡ (¬C? ∧ C! ∧ stable(C) ; skip) ∨ empty ; C.e
C?x ≡ (¬C! ∧ C? ∧ stable(C) ; skip) ∨ empty ; C.x
```

The projection function Π₁ gives the ”willing to read” value, while the projection function Π₂ gives the ”willing to write” value. The projection function Π₃ has the actual value in the channel. The notations *C!*d*e* (C?d*x* formally describes that an agent is willing to perform the communication at time *d* where *d* ∈ *TIME*. However, in case the environment fails to react promptly, the system will be on hold forever:

```
C!d*e ≡ C!e ∧ (finite ⊃ len = d)
C?d*x ≡ C!x ∧ (finite ⊃ len = d)
```
3.2.1.4 Shunt Communication

The tuples \((t, v)\) represent the values of the shunt \(s\) variables, where \(t\) is a stamp and \(v\) is the written value. To introduce a shunt \(s\), shunt \(s\) in \(P\) is used. To denote the written value \(v\) to shunt \(s\), I used write\((v, s)\). To read the stored value in shunt \(s\), I used read\((s)\). To give the stamp of shunt \(s\), I used \(\sqrt{s}\):

\[
\begin{align*}
\sqrt{s} &\triangleq \Pi_1(s) \\
\text{shunt } s \text{ in } P &\triangleq \exists s \cdot \sqrt{s} = 0 \land P \\
\text{write}(v, s) &\triangleq \text{skip} \land o s = (\sqrt{s} + 1, v) \\
\text{read}(s) &\triangleq \Pi_2(s)
\end{align*}
\]

The projection function \(\Pi_1\) gives the stamp while the projection function \(\Pi_2\) gives the value stored in shunt \(s\). The notation \(\text{write}_d(v, s)\) formally describes an agent that writes to shunt \(s\) the value \(v\) at time \(d\) where \(d \in Time - \{0\}\):

\[
\text{write}_d(v, s) \triangleq len = 1 - 1 \land \text{skip} \land o s = (\sqrt{s} + 1, v)
\]

In case the agent \(\text{write}_d(v, s)\) is required to stay stable except of the last state of the interval, the agent \(\text{pwrite}_d(v, s)\) takes over as follows:

\[
\begin{align*}
\text{pwrite}_d(v, s) &\triangleq \text{write}_d(v, s) \land \text{padded}(s) \\
\text{padded}(s) &\triangleq (\text{stable}(s); \text{skip}) \lor \text{empty}
\end{align*}
\]

where \(\text{padded}(s)\) is a padded expression, and it has been formally defined in Interval Temporal Logic (ITL) as shown the above formula.
CHAPTER 3. COMPUTATIONAL MODEL

3.2.1.5 Delay and Timeout

The notation $\text{delay}_d$ formally describes an agent that sets on hold at first for $d$ time units, where $d \in \text{TIME} \cup \{\infty\}$, and then it gets terminated without updating the global variables:

$$\text{delay}_d \equiv \text{len} = d$$

The notation $P \preceq_d Q$ formally describes an agent behaviour such as $P$ if $P$ is executed within $d$ time units, otherwise agent $Q$ takes over the execution:

$$P \preceq_d Q \equiv \text{if } (P \supset \text{finite} \land \text{len} \leq d) \text{ then } P \text{ else } Q$$

3.2.1.6 Resource Allocation

The $v$ units of resource $res$ can be requested via the agent $\text{request}(v, res)$. The agent waits for $v$ units in case they are not available [47]. The agent $\text{release}(v, res)$ is used to release $v$ units of the resource $res$:

$$\text{request}(v, s) \equiv \text{if } res \geq v \text{ then } res := res - v \text{ else } \oslash(\text{request}(v, res))$$

$$\text{release}(v, s) \equiv \oslash res = res + v$$

3.2.1.7 The Funnel

A restricted form of multiplexing on shunts can be performed using the agent called funnel. The syntax of the agent funnel is $s_i \rightsquigarrow_I s_{out}$ describes the connection of shunts $s_i$, which is indexed by $i$, to the shunts $s_{out}$. When a write operation occurs in shunts $s_j$ where $j \in I$ then shunts $s_{out}$ must have a write operation at the same time. Shunts $s_i$ and $s_{out}$ stay stable if no write operation
occurs. The funnel becomes false when two different values are written to shunts $s_i$ and $s_j$ at the same time:

\[
\begin{align*}
s_i \leadsto v_s & \triangleq (\bigwedge_{i \in I} \text{stable}(s_i) \land \text{stable}(s_{\text{out}})) \lor \\
((\bigvee_{i \in I} \text{stable}(s_i)) \land \text{skip} \land \sqrt{s_i} := \sqrt{s_i + 1}) \land \\
\exists v, t \cdot len = t \land \\
((\bigwedge_{i \in I} \text{stable}(s_i)) \land \text{skip} \land \sqrt{s_i} := \sqrt{s_i + 1} \cup \text{fin}(\text{read}(s_i)) = v) \land \\
\text{pwrite}_t(v, s_{\text{out}})
\end{align*}
\]

According to Cau and Zedan [47], the funnel allows to execute agents concurrently to the same shunt with making the assumption of no conflict is occurring. As an agent may perform reading and writing to shunts, it requires at least two time units to update the stamp.

3.2.2 Shared-Variable based Communication

Shared-variable is a model of communication between parallel systems that share a variable. All systems can read and write to the variable whenever they need to. There are constraints on shared-variable model to ensure consistency of the value of shared variable among all systems that share it:

- All parallel systems can read the consistent value of the shared variable at the same time (Concurrent Read CR).

- Only one system can write to the shared variable at a time (Exclusive Write EW). Mutual Exclusion mechanism is applied to ensure this constraint. For instance, if $Sys_1$ needs to write to a shared variable $Data$, a lock-based solution is used to enforce a Mutual Exclusion synchronisation mechanism on a shared variable as the following:

\[
\text{Lock}(Data); \quad Data = x; \quad \text{Unlock}(Data);
\]
Only one system, for instance $Sys_1$, is allowed to write the value $x$ to a shared variable $Data$ at a time. The above two constraints use one model of the Parallel Random Access Machine (PRAM) models, which is a Concurrent Read Exclusive Write (CREW) [275].

### 3.2.3 True Concurrency

True concurrency form allows the parallel systems to be independently executed at the same time. If parallel systems share a variable, then a synchronisation mechanism such as mutual exclusion has to be applied to ensure a consistent value of a shared variable. Figure 3.3 illustrates this form of concurrency where $Sys_1$ and $Sys_2$ represent any system running in parallel, and $\sigma_n$ represents the state number. A global state construction for parallel systems which run in true concurrency is defined in Definition 9 and illustrated in Figure 3.4.

![Figure 3.3: Parallel Composition of $Sys_1$ and $Sys_2$ (True Concurrency)](image)

**Definition 9** Global State Construction (True Concurrency): The locals state numbers $i$ and $j$ of parallel systems, $Sys_1$ and $Sys_2$, which run by the local processors $n$ and $m$ respectively are equivalent, while the global state number of the composed systems, $Sys_1$ and $Sys_2$, which run by the global processor $g$ is $k$; $k$ is the sum of the locals state numbers divided by $x$, the number of available processors, as follows:

$$\sigma^n_i \parallel_T \sigma^m_j \equiv \sigma^g_{k=(i+j)/x}$$
CHAPTER 3. COMPUTATIONAL MODEL

where:

$\|_T$ is the parallel (true concurrency) operator symbol,

$\sigma^n_i$: state number $i$ of processor $n$,

$\sigma^m_j$: state number $j$ of processor $m$,

$g$: global processor,

$k$: global state number.

\[
\sigma^1_0 \parallel_T \sigma^2_0 \equiv \sigma^g_0 \quad \sigma^1_1 \parallel_T \sigma^2_1 \equiv \sigma^g_1 \quad \sigma^1_2 \parallel_T \sigma^2_2 \equiv \sigma^g_2 \quad \sigma^1_3 \parallel_T \sigma^2_3 \equiv \sigma^g_3 \quad \sigma^1_4 \parallel_T \sigma^2_4 \equiv \sigma^g_4
\]

Figure 3.4: Global State Construction (True Concurrency)

3.2.4 Interleaving Concurrency

Interleaving concurrency allows only one system to be executed at a time. When one system is running, the other parallel systems are idle. Synchronisation mechanism is not required in this form because there is no concurrent writes to a shared variable and any change gets committed at every state, which allows the other parallel systems to see the updates in the next state. Figure 3.5 illustrates this form of concurrency where $Sys_1$ and $Sys_2$ represent any system running in parallel, and $\sigma^n$ represents the state number.
A global state construction for parallel systems which run in interleaving concurrency is defined in Definition 10 and illustrated in Figure 3.6.

**Definition 10 Global State Construction (Interleaving Concurrency):** The locals state numbers $i$ and $j$ of parallel systems, $Sys_1$ and $Sys_2$, which run by the local processors $n$ and $m$ respectively are inequivalent, while the global state number of the composed systems, $Sys_1$ and $Sys_2$, which run by the global processor $g$ is $k$; $k$ is the sum of the active processor’s local state number and the stuttered processors state numbers as follows:

$$
\sigma^n_i \parallel_I \sigma^m_j \equiv \sigma^g_{k-i+j}
$$

where:

- $\parallel_I$: parallel (interleaving concurrency) operator symbol,
- $\sigma^n_i$: state number $i$ of processor $n$,
- $\sigma^m_j$: state number $j$ of processor $m$,
- $g$: global processor,
- $k$: global state number.
3.3 Architecture Framework

As stated in section 3.1, Parallel Runtime Verification Framework (PRVF) has different levels and phases. The levels are global and locals, while the phases are generation phase, local verification & assertion phase, and global verification phase. The global level includes the generation phase in the beginning and the global verification phase in the end of the architecture of the framework. The locals level includes the locals verification & assertion phase in the middle of the architecture of the framework. These levels and phases are illustrated in Figures 3.7 to 3.10.

3.3.1 Generation Phase

This phase lies within the global level of a framework where the global level is intended to generate processor identification $\text{Pid}$, global data $\text{Data}$, communication model $\text{Communication}$, concurrency form $\text{Concurrency}$, and execution mode $\text{Execution}$. The generation of $\text{Pids}$ occurs randomly via the modulo operation where a random number gets modulo over the available number of the processors that run in parallel. Data is generated according to the nature of the
system under scrutiny.

### 3.3.1.1 Communication Models

The value of *Communication* determines the mechanism of the communication between parallel systems. When the value is 0, then the mechanism of the communication is *shared-variable*. When the value is 1, then the mechanism of the communication is *message-passing*.

### 3.3.1.2 Concurrency Forms

The value of *Concurrency* is assigned to indicate the concurrency form which is either interleaving or true concurrency (0 for interleaving, 1 for true concurrency). The execution value determines the execution mode of systems running parallel in terms of communication.

### 3.3.1.3 Execution Modes

The execution mode is either synchronous (the value 0 is set) or asynchronous (the value 1 is set). For synchronous execution of the communication, I used *Channels*, while for asynchronous execution of the communication, I used *Shunts*.

These generated data then get broadcast to all local parallel systems with $\text{Pid} = 0$ to $K - 1$, where $K$ is the number of available processors that are intended to run systems in parallel with all possible paths which might be encountered.

Algorithm 1 defines the mechanism that is used in PRVF model to generate *Pid* and *Data* randomly, and set values for a communication mechanism, concurrency form, and execution mode. The communication value determines which mechanism is being used. For instance, the *Communication*’s value 0 sets the mechanism to shared-variable, while the *Communication*’s value 1 sets the mechanism to message-passing. Within each communication mechanism there are varieties of concurrency forms. When the value of *Concurrency* is 0, then concurrency is interleaving. At every single cycle, one local candidate wins the assignment to be the active processor, while the others set to be idle. When the value of *Concurrency* is 1, then the concurrency is true concurrency which means that at every single cycle all locals become active. The execu-
CHAPTER 3. COMPUTATIONAL MODEL

Algorithm 1: Generation Phase

1. **Generate**\( (\text{Pid}, \text{Data}, \text{Concurrency}, \text{Execution}) \);
   
   **Input**: Communication, Concurrency, Execution
   
   **Output**: Pid, Data

2. for \( i \leftarrow 0 \) to \( n \) do

3. \( \text{Pid} = \text{Random \ mod \ } K; \) \hspace{1cm} \( K \) IS THE NUMBER OF PROCESSORS.

4. \( \text{Data} = \text{Random \ mod \ } N; \) \hspace{1cm} \( N \) IS ANY NATURAL NUMBER.

5. \( \text{Communication} = \text{Random \ mod \ } 2; \) \hspace{1cm} COMMUNICATION IS EITHER 0 TO SET IT TO SHARED-VARIABLE, OR 1 TO SET IT TO MESSAGE-PASSING.

6. if Communication = 0 then

7. \( \text{Concurrenty} = \text{Random \ mod \ } 2; \) \hspace{1cm} CONCURRENTY IS EITHER 0 TO SET IT TO INTERLEAVING, OR 1 TO SET IT TO TRUE CONCURRENTY.

8. else

9. \( \text{Execution} = \text{Random \ mod \ } 2; \) \hspace{1cm} EXECUTION IS EITHER 0 TO SET IT TO SYNCHRONOUS, OR 1 TO SET IT TO ASYNCHRONOUS.

10. end

11. foreach \( j \leftarrow 0 \) to \( K - 1 \) do

12. if Communication = 0 then

13. \( \text{Send}(j, \text{Pid}, \text{Data}, \text{Concurrency}); \)

14. else

15. \( \text{Send}(j, \text{Pid}, \text{Data}, \text{Execution}); \)

16. end

17. end

18. end

The execution mode has two values, either 0 or 1. When the value of Execution is 0, then the execution mode is synchronous; otherwise, it is asynchronous.

Shared-variable based communication is associated with the concurrency forms such as interleaving or true concurrency. However, the association between execution modes and shared-variable based communication is omitted. On the other hand, message-passing based communication is associated with the execution modes such as synchronous or asynchronous. However, the association between concurrency forms and message-passing based communication is also omitted. The reason for these two omissions is because the proposed model demonstrates all the
possibilities of communication versus all the possibilities of concurrency of parallel systems. In other words, the applicability of these approaches can be tailored accordingly.

3.3.2 Locals Verification & Assertion Phase

Locals Verification & Assertion phase lies within the locals level of the framework. The Locals level is intended to synchronise the execution of parallel systems according to the data that are sent from the global level. Shared-variable communication mechanism is determined via the assignment of communication value. Algorithm 1 describes and models all the possibilities that the proposed framework might encounter.

3.3.2.1 Interleaving Concurrency and Shared-Variable

I assume that the value of Communication is 0 which means that the communication mechanism is shared-variable. When the value of Concurrency is 0, all locals receive these data and compare the received Pid from the global level with their local Pids. If a local matches its Pid with the received Pid from the global, it precedes; otherwise, a local sets itself to idle. Then, the received global data gets assigned to local data DataL to be locally checked against a property of interest Propi.

Algorithm 2: Locals Verification & Assertion Phase of Processor i (Interleaving)

1. Assert(Propi);
2. Input : Pid, Data, Concurrency
3. Output: Propi
4. if Communication = 0 ∧ Concurrency = 0 then
5.     if Pid = i then
6.         DataL = Data;
7.         Check(Propi);
8.         Assert(i, Propi);
9.     else
10.        Print ”Local i is Idle”;
11.     end
12. else
13.     Exit();
14. end
CHAPTER 3. COMPUTATIONAL MODEL

Figure 3.7: Parallel Runtime Verification Framework (Shared-Variable Interleaving Concurrency)
CHAPTER 3. COMPUTATIONAL MODEL

After that, the local property $\text{Prop}_i$ is asserted to be sent to the global level and the global verification phase (see Algorithm 2 & Figure 3.7).

3.3.2.2 True Concurrency and Shared-Variable

I assume that the value of $\text{Communication}$ is 0 which means that the communication mechanism is shared-variable. When the value of $\text{Concurrency}$ is 1, the fastest local system locks the received data $\text{Data}$ to be able to exclusively write to it.

Once the write operation is done, $\text{Data}$ gets unlocked and synchronised with all the locals to enforce data consistency of shared variable $\text{Data}$. Then, a property $\text{Prop}_i$ is checked locally against a set of specifications within all locals system that are interested in the shared variable $\text{Data}$. After that, the locals’ properties $\text{Prop}_i$ and $\text{Prop}_j$ are asserted to be sent to the global level and the global verification phase. Algorithm 3 describes this model.

Algorithm 3: Locals Verification & Assertion Phase of Processor $i$ (True Concurrency)

1. Assert($\text{Prop}_i$);
   
   **Input**: $\text{Pid}, \text{Data}, \text{Concurrency}$
   
   **Output**: $\text{Prop}_i$

2. if $\text{Communication} = 0 \land \text{Concurrency} = 1$ then
   3. Lock($\text{Data}$);
   4. $\text{Data}_L = \text{Data}$;
   5. Unlock($\text{Data}$);
   6. Sync($\text{Data}$);
   7. Check($\text{Prop}_i$);
   8. Assert($i, \text{Prop}_i$);

9. else
10. Exit();
11. end

Figure 3.8 illustrates the components of this model including levels and phases. The Levels are global and locals, while the phases are generation, assertions and locals verification, and global verification. The flowchart visually describes the flow of the data within this model. Flowcharts components such as process and decision are primarily used to describe this version of the model.

58
CHAPTER 3. COMPUTATIONAL MODEL

Figure 3.8: Parallel Runtime Verification Framework (Shared-Variable True Concurrency)
CHAPTER 3. COMPUTATIONAL MODEL

3.3.2.3 Synchronous Execution and Message-Passing (Channels)

The message-passing communication mechanism is encountered when the value of Communication is 1. When a communication mechanism is shared-variable, only a concurrency form has to be set and sent to locals level. On the contrary, when a communication mechanism is message-passing, only an execution mode has to be set and sent to locals level. Being within message-passing communication mechanism implies that the value of Communication is 1. Figure 3.9 & Algorithm 4 illustrate this model. The first decision process is encountered within locals level for the selected path is the execution mode. There are two execution modes which are either synchronous or asynchronous. Synchronous execution of message-passing communication mechanism is modelled using a construct called Channel communication.

<table>
<thead>
<tr>
<th>Algorithm 4: Locals Verification &amp; Assertion Phase of Processor i (Synchronous Message-Passing (Channels))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Assert(Prop_i);</td>
</tr>
<tr>
<td>Input : Pid, Data, Concurrency</td>
</tr>
<tr>
<td>Output: Prop_i</td>
</tr>
<tr>
<td>2 if Communication = 1 ∧ Execution = 0 then</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17 else</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19 end</td>
</tr>
</tbody>
</table>
Figure 3.9: Parallel Runtime Verification Framework (Synchronous Message-Passing)
CHAPTER 3. COMPUTATIONAL MODEL

Channel construct has triple of these values \( wtr, wtw, \) and \( v \). The first one, \( wtr \), is a boolean value, and its value indicates whether the system is willing to accept (read) a message from that channel. The second one, \( wtw \), is a boolean value, and its value indicates whether the system is willing to send (write) a message to that channel. The third one, \( v \), stands for the value currently in channel \( C \) when \( wtr \) and \( wtw \) are both true.

When \( wtr \) is true, \( \text{Local}_i \) reads what is being passed through the channel \( C \) at time \( d \). When \( wtw \) is also true, \( \text{Local}_i \) writes the value \( v \) to the channel \( C \) at time \( d \). Once the write operation is done, the written value \( v \) is checked to determine whether it is satisfying the desired property \( \text{Prop}_i \) of \( \text{local}_i \). Then, a local property \( \text{Prop}_i \) is asserted to be delivered to the global verification phase.

3.3.2.4 Asynchronous Execution and Message-Passing (Shunts)

Back to the first decision process encountered within locals level for the selected path which is an execution mode, the second execution mode is asynchronous execution of message-passing communication mechanism which is modelled using a construct called \textit{Shunt} communication. Algorithm 5 illustrate this model.

Shunt construct has tuple of these values \( t \), and \( v \), where \( t \) is a stamp and \( v \) is the written value. The construct shunt \( s_i \) belongs to \( \text{Local}_i \) uses the write agent \( \text{write}_d(v, s_i) \) to denote that at time \( d \), shunt \( s_i \) has the value \( v \) written to it. The read agent \( \text{read}_d(s_i) \) denotes the value stored in shunt \( s_i \). The stamp agent \( \sqrt{s} \) denotes the stamp of the shunt \( s_i \).

The funnel allows agents to write concurrently at the same time to the same shunt \( s \). When shunt \( s \) has different values written to it via different agents e.g. \( i \) and \( j \) at the same time \( d \), the funnel becomes false. When the agents \( i \) and \( j \) write the same value at the same time time \( d \), the write operation occurs instantly in \( s_{out} \). The written value is then checked against a desired property. After that, the local \( \text{Prop}_i \) gets asserted to be sent to the global verification phase.
Algorithm 5: Locals Verification & Assertion Phase of Processor $i$ (Asynchronous Message-Passing (Shunts))

1. **Assert**($Prop_i$);

   **Input**: $Pid, Data, Concurrency$

   **Output**: $Prop_i$

2. **if** $Communication = 1 \land Execution = 1$ **then**
   3. $write_d(v, s_i)$;
   4. $read_d(s_i)$;
   5. $stamp(s_i)$;
   6. $Send(j, s_i)$;
   7. $Receive(j, s_j)$;
   8. **if** $s_i = s_j$ **then**
      9. $s_{out} = s_i$;
      10. **Check**($Prop_i$);
      11. **Assert**($i, Prop_i$);
   12. **else**
      13. **Print** "The Funnel is false because shunts $i$ & $j$ wrote different values at the same time";
   14. **end**
  15. **else**
  16. **Exit();**
  17. **end**

Figure 3.10 illustrates the components of this model including levels and phases. The Levels are global and locals, while the phases are generation, assertions and locals verification, and global verification.
CHAPTER 3. COMPUTATIONAL MODEL

Figure 3.10: Parallel Runtime Verification Framework (Asynchronous Message-Passing)
CHAPTER 3. COMPUTATIONAL MODEL

3.3.3 Global Verification Phase

This phase lies within a global level. At this phase locals’ properties are received from the locals assertion phase. The global verification phase gets locals’ properties in order to compose a global property out of the received locals’ properties as Algorithm 6 illustrates. When the concurrency form is interleaving concurrency, \( Concurrency = 0 \), then only one local property is gotten due to the concurrency form. The property of the active local is received. When the concurrency form is true concurrency, \( Concurrency = 1 \), then all locals properties are gotten due to the concurrency form. The properties of interest of all locals are received.

Message-passing communication models can be handled according to the concurrency form being used. I omit the concurrency forms for message-passing due to fact that my interest is to show all the possible models without redundancy, for instance, true and interleaving concurrency are demonstrated in association with shared-variable communication mechanism; therefore, no need to demonstrate it in association with message-passing communication mechanism. The same idea applies to (a)synchronous execution modes.

\[
Local_i \preceq_d Local_j \equiv \text{if } (Local_i \supset finite \land \text{len} \leq d) \text{ then } Local_i \text{ else } Local_j
\]
CHAPTER 3. COMPUTATIONAL MODEL

Timeout agent deals with locals which do not behave in a time manner. For instance, $Local_i$ is expected to do its task within $d$ time units. $Local_j$ takes over, otherwise.

3.4 Parallel Runtime Verification Framework (PRVF) Model

Parallel Runtime Verification Framework (PRVF) model allows the collection of requirements from several sources to handle local and global correctness properties. The model also allows sending and receiving assertion data from several sources to handle true/interleaving concurrency associated with shared-variable based communication approach. In addition, the model enables the application of mutual exclusion synchronisation mechanism and the use of lock-based technique in order to guarantee synchronised and consistent shared variables.

Parallel Runtime Verification Framework (PRVF) model allows handling synchronous/asynchronous communication links such as Shunts/Channels associated with message-passing based communication approach. The model offers the ability to execute agents concurrently via the funnel besides the introduction of resource allocation agents request and release.

Parallel Runtime Verification Framework (PRVF) model introduces Delay ($Delay$) and Timeout ($P \leq_d Q$) agents which play an important role in managing such a behaviour. It also offers checking the correctness properties of local systems at the locals and global levels. Consequently, inference of the correctness global property can be derived from the correctness of a set of local properties of global systems. These new capabilities are demonstrated in the next chapter, Chapter 4.

3.5 Summary

In this chapter, the computational model, namely, Parallel Runtime Verification Framework (PRVF) is introduced. Communication mechanisms such as shared-variable and message-passing are identified. Concurrency forms such as true concurrency and interleaving concurrency are identified as they are intended to be used in the proposed model. Additionally, PRVF can handle synchronous execution of message-passing via a construct called channel and asynchronous
execution of message-passing via a construct called *shunt*. A comprehensive description of the components and capabilities of PRVF is given. In the next chapter, the implementation of PRVF model is demonstrated.
Chapter 4

Design and Implementation of a Parallel Runtime Verification Framework (PRVF)

Objectives:

- To review the current version of AnaTempura
- To describe the Development of Parallel Runtime Verification Framework (PRVF) model
- To show the Implementation of PRVF model using Java, Tempura, and AnaTempura
- To highlight the Impact of PRVF model on AnaTempura Evolution Aspects
- To demonstrate Benchmarking Applications using PRVF model
4.1 Introduction

In this chapter, the computational model, namely, Parallel Runtime Verification Framework (PRVF) is designed and implemented. The proposed model is an extension of a runtime verifier tool called AnaTempura. First, a description of the current model of AnaTempura is reviewed in order to address the drawbacks of AnaTempura model. After that, the proposed model of a Parallel AnaTempura is represented and a demonstration is given to show how it bridges the gaps for parallel systems. Benchmarking applications such as Producer-Consumer and Dining Philosophers Problem are implemented using the proposed model.

4.2 (Ana)Tempura

AnaTempura is a runtime verifier of systems using Interval Temporal Logic (ITL) and its executable subset Tempura. It uses assertion points as a technique at runtime verification to check system satisfaction or violation of a property of interest such as timing, safety, security which are formally expressed in Interval Temporal Logic (ITL).

Figure 4.1: General System Architecture of AnaTempura [301]
The assertion points get inserted in the source code of a system under scrutiny and subsequently a sequence of information such as variables’ names and their values, timestamps values are generated. The generated data then get checked against the expected values that match a property of interest.

A property is an expected behaviour of a system over a sequence of states (interval). The property gets expressed in Interval Temporal Logic (ITL) and then modelled in Tempura language to get it executed and checked against that property. AnaTempura does this membership test as it has Tempura interpreter and the monitor [52, 54]. The main components of AnaTempura are illustrated in Figure 4.1. A description of AnaTempura’s main components including Assertion Points, The Monitor, and Tempura Interpreter is given in the next sections.

AnaTempura is a semi-automatic tool which means a human intervention is unavoidable due to the complexity to understand systems automatically. The integration between Interval Temporal Logic (ITL) and its executable subset Tempura allows AnaTempura to offer:

- Formal specification

- Validation and verification of a formal specification throughout simulation and runtime checks

![Desired Properties (Tempura Code) → Validate → System (Source Code + Assertion points)](image)

Figure 4.2: The Analysis Process [301]

The analysis process as illustrated in Figure 4.2 checks the system’s source code in addition to the assertion points within it against the desired properties modelled and written in Tempura language. The source code of a system could be written in C, C#, Java, Scala, Verilog, or Tempura.
4.2.1 Assertion Points

Assertion points is a mechanism that enables systems engineer/analyst to gather information within a source code of these systems to analyse their behaviour over time. Assertion points get asserted after every state which is a mapping between variables and their values. A set of variables which is used to express the property of interest has to be determined. After that, the assertion points get inserted directly after the value assignment to these variables. Figure 4.3 illustrates assertion points general mechanism where $B_1$ and $B_2$ are the assertion points to reflect the change of code chunk of $C_1$.

![Figure 4.3: Assertion Points and Chunks](image)

Assertion points generate data which reveal information at runtime about a system under scrutiny. This information includes States and Time Stamps:

- States information maps between the variables that express a property and their values. This mapping technique has the format $\langle \text{Var}, \text{Val} \rangle$, for instance:

  $$\langle \text{Pid}, 1 \rangle \langle \text{RW}, 0 \rangle \langle \text{Addr}, 3 \rangle$$

where three variables $\langle \text{Pid}, \text{RW}, \text{Addr} \rangle$ and their values $\langle 1, 0, 3 \rangle$ are inserted respectively. The inserted variables represent a processor identification $\text{Pid}$, Read or Write operation $\text{RW}$, Memory Address $\text{Addr}$. These variables are part of the cache controller case study which is intended to be studied in Chapter 5. The above assertion point reveals that a cache controller system creates a request to read ($\text{RW} = 0$ read, $\text{RW} = 1$ write) a memory address.
CHAPTER 4. DESIGN AND IMPLEMENTATION OF A PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

Memory[Addr] and this request is assigned to a processor which has a Pid value 1. This generation of information can then reveal and check whether a system’s behaviour is either satisfying or violating a certain property which has to be met.

- Time Stamps information maps between different assertion points where variables and their values within these assertion points are changed and to record at what time a change has occurred. A system’s clock is used to obtain time stamps. In addition to variable and values parameters, a time stamp parameter is included to form sets of triples instead of pairs. The triple format is \((\text{Var}, \text{Val}, \text{Time Stamp})\), for instance:

\[
\langle \text{Pid}, 1, 8 \rangle \langle \text{RW}, 0, 8 \rangle \langle \text{Addr}, 3, 8 \rangle
\]

\[
\langle \text{Pid}, 1, 9 \rangle \langle \text{RW}, 0, 9 \rangle \langle \text{Addr}, 3, 9 \rangle
\]

where the assertion points add a time stamp value to show a change of the asserted data between time unit 8 and 9. Time stamps could be in microseconds, seconds, minutes, hours etc. When a memory address, Memory[3], has changed its value within these time stamps, then a judgement in regards of a property of interest can be made.

The determination of a location and number of assertion points within a source code is still manual and relies on systems engineer/analyst’s understanding of a system under scrutiny [300]. The mechanism of capturing and interpreting assertion points is illustrated in Figure 4.4. There are two components which are intended to receive assertion data generated by assertion points within a source code of a system, and then split them accordingly into three groups.

The groups as the figure illustrates are variable name, value, and time stamps. The first component is Data Capture, and it captures the assertion data as strings and then forwards them
to Data Interpret component. The string has the following format:

```
!PROG: assert variable name: value: time stamp: !
```

The above clause has a set of markers. Each marker has a meaning as follows:

“!PROG” This marker indicates that assertion data are generated from a program.

“assert” indicates the data being asserted.

“:” The colon symbol separates the asserted data.

“!” The exclamation symbols terminates the assertion data clause.

Based on these markers, a Data Interpret component divides the strings into three groups which are variable name, value, and time stamps. Then these assertion data are sent to Tem-
pura interpreter in order to execute them and then send the corresponding output to the monitor. Listing 4.1 illustrates how assertion points look like within a Java external program.

Listing 4.1: Generating Assertion Points within Java Program

```java
class AssertionPoints {
    public static void main(String[] args) {
        int Pid, RW, Addr, Timestamp;
        Pid = 1; RW = 0; Addr = 3; Timestamp = 9;
        System.out.println("!PROG: assert Pid:" + Pid + ":" + Timestamp + ":!");
        System.out.println("!PROG: assert RW:" + RW + ":" + Timestamp + ":!");
        System.out.println("!PROG: assert Addr:" + Addr + ":" + Timestamp + ":!");
    }
}
```

The assertion points in line 5, 6, and 7 within Listing 4.1 inserts three variables names and their values in addition to the time stamp’s value. The variable set is $\langle \text{Pid}, \text{RW}, \text{Addr} \rangle$, while the value set of these variables is $\langle 1, 0, 3 \rangle$ respectively to their variables names in addition to the time stamp value which is 9. The external Java program represents a system to analyse. AnaTempura allows systems to be plugged-in with Tempura interpreter via a monitor. To associate an external program with a Tempura file, line 3 within Listing 4.2 has to be placed here. Figure 4.5 illustrates a successful compilation of an external Jave program via AnaTempura which is plugged in to a Tempura program. Once a Java external program is executed, a string of assertion data is sent to

![Figure 4.5: Compiling External Java Program](image-url)
Tempura program after they get captured and interpreted accordingly. Tempura has a mechanism that allows the assertion data to be assigned to a list of variables within the Tempura program intended to be checked via specific functions. These functions are listed in lines 4, 5 and 6 within Listing 4.2. The function in line 4 is intended to pass variable names. The function in line 5 is intended to pass values of those variables, while the functions in line 6 are intended to pass the time stamps in seconds. These functions allow us to pass the assertion data through them and assign the received values to internal variables to be deployed internally.

Listing 4.2: Collecting Assertion Data within Tempura Program

```plaintext
load "../library/conversion".
load "../library/exprog".
/* java AssertionPoints 0 */
define avar(X) = {X[0]}.
define aval(X) = {X[1]}.
define atime(X) = {strint(X[2])}.
set print_states = true.
define get_var(Variable,Value,Timestamp) = {
    exists T : {
        get2(T) and
        Variable = avar(T) and
        Value = strint(aval(T)) and
        Timestamp = atime(T) and
        format("Assertion data <%s, %d, %d> are received!
",Variable,Value,Timestamp)
    }
}.
/* run */ define Test() = {
    exists Variable,Value,Timestamp : {
        get_var(Variable,Value,Timestamp) and len(0); skip;
        get_var(Variable,Value,Timestamp) and len(0); skip;
        get_var(Variable,Value,Timestamp) and len(0)
    }
}.
```
Once Tempura runs a test in line 17 within Listing 4.2, the monitor shows the assertion data imported to the test. The assertion data, which has been asserted within an external Java program, are successfully printed out within the monitor as Figure 4.6 illustrates.

![Figure 4.6: RUNNING TEMPURA PROGRAM](image)

**4.2.2 The Monitor**

The monitor is a user-friendly interface which has been built to be an interactive system by allowing system engineers/analysts to insert inputs during the runtime in order to be able to analyse the time-critical systems. The monitor is responsible for capturing and analysing the assertion data which is generated by assertion points. Based on a set of criteria set by system engineers/analysts, the monitor can make a judgement on a system behaviour against properties such as safety, liveness, and projected time. The monitor has a textual interface and graphical interface. The Tcl/Tk [288] and Expect [155] were initially used to build the tool. Tcl/ Tk graphical user interface no longer depend on Expect, this has been the case since the release of version 3.3 of AnaTempura. The latest release of an up-to-date AnaTempura is version 3.4. [54].

When the run of a system initialised within the monitor, the assertion points which are placed within a source code of a system under scrutiny send their assertion data to the monitor. The monitor then receives these assertion data accordingly and send them to Tempura interpreter. The Tempura interpreter then checks the executable specifications written in Tempura file against the received assertion data and after that a judgement of pass or fail is made accordingly. The
Tempura interpreter indicates to the failure’s location and explains why the failure occurred. This information is displayed via the monitor.

### 4.2.3 Tempura Interpreter

Tempura interpreter is an interpreter of executable Interval Temporal Logic formulae. The current Tempura interpreter is programmed in C language and denoted as C-Tempura. The C-Tempura interpreter was originally developed by Roger Hale in 1985 at Cambridge University, and now it is maintained by Antonio Cau and Ben Moszkowski. However, Ben Moszkowski developed the first Tempura interpreter, and it was programmed in Prolog in December 1983. In March 1984, Ben Moszkowski rewrote the interpreter in Lisp [54]. I refer the reader to Moszkowski’s book [182] for more details.

### 4.3 Evolutionary Improvements of AnaTempura

A single vending machine can serve one person at a time. When there are ten people queuing to be served in order to get hot beverages and while each beverage consumes 10 seconds to be delivered, the total needed time to serve ten people is 100 seconds. But when there is another vending machine, half of the load on the first machine is transferred to the second machine, which means five people would be queuing at each vending machine. The existence of the other vending machine reduces the load to the half and consequently the consumed time is as well reduced to 50 seconds to serve all the ten people. This significant reduction of the consumed time is due to the speed increment with the assumption of having 100% parallel portion (100% = 1, 50% = 0.50) and two vending machines. Amdhal’s Law [115] is used to perform the calculation of the speed for parallel computation. Amdhal’s Law is defined as the following:

\[
    \text{Speedup}(N) = \frac{1}{(1-P) + \frac{P}{N}}
\]

(4.1)
where $P$ is a parallel portion of a system in percentage; $N$ is the number any kind of objects that are intended to perform parallel tasks, for instance vending machines. The application of Amdhal’s Law assumes that the speed is exponentially incremented in accordance with the number of available parallel processes in execution which consequently leads to significant improvement in performance. Applying Amdhal’s Law on vending machines’ example produces the following result:

$$\text{Speedup}(2) = \frac{1}{(1-1)+\frac{1}{2}} = \frac{1}{0.50} = 2\ times$$

The speed is doubled which means only half of the time is needed to perform the task. Instead of consuming 100 seconds at one vending machine, only 50 seconds are needed when there are two vending machines. Amdhal’s Law defines the incremental relationship between the number of processors and the performance as illustrated in Figure 4.7.

![Amdahl's Law](image)

Figure 4.7: AMDAHL’S LAW [115]

Dividing the ten people into two groups, and with each group consisting of five people to be served by only one machine in interleaving concurrency form of parallelism does not change the fact that serving them sequentially as one group of ten leads to the same result of serving them in
interleaving concurrency form of parallelism. Therefore, practically, sequential and interleaving concurrency mechanisms are alike in terms of performance. Performance increases significantly by applying true concurrency form of parallelism. True concurrency form needs parallel software/hardware components and a channel of communications in case of shared resources.

The current version of AnaTempura can not handle parallel systems at a time because the current framework as illustrated in Figures 4.1 and 4.8 has single components such as The Monitor (The Server) and Tempura Interpreter. The single monitor can only monitor one system at time; also, the single Tempura Interpreter can execute only one Tempura program at a time and this is the same for the rest of the components. Therefore, multiple components are needed to handle parallel systems at a time. The proposed model has tackled this issue by deploying and introducing the principles of parallelism to AnaTempura to enable it to handle all forms of parallelism at a time and architectures such as Multi-cores/processors, Parallel Random Access Memory (PRAM), and Remote Method Invocation (RMI).
4.3.1 Realisation of Assertion Points Techniques

In this section, a set of realisation of assertion points techniques of the proposed model, Parallel Runtime Verification Framework (PRVF), are introduced and explained in details.

- The variety of source of the requirements that handle local and global properties implies the collection of assertion data from several sources, at a time, to handle concurrency. In addition to the multiple assertion points within several sources, the assertion points clause is extended in order to allow more variables and values to be asserted at a time. The extended format is as follows:

\[
\langle \text{Pid}_{\text{var}}, \text{Pid}_{\text{val}}, \text{Var}_n, \text{Val}_n, \ldots, \text{Var}_m, \text{Val}_m, \text{Timestamp} \rangle
\]

where \(\text{Pid}_{\text{var}}\) could be program, process, or thread identification number, \(\text{Pid}_{\text{val}}\) is the value of \(\text{Pid}_{\text{var}}\), \(\text{Var}_n\) is the \(n^{th}\) variable, \(\text{Val}_n\) is the \(n^{th}\) value of \(n^{th}\) variable, and \(\text{Timestamp}\) is a time stamp of the assertion points where time now can be in microseconds.

Listing 4.3 illustrates the extended assertion points in correspondence to the functions introduced in Listing 4.4 to allow more variables and values to be asserted at a time and collected at once.

Listing 4.3: Generating Assertion Points within Java Program

```java
class ExtendedAssertionPoints {
    public static void main(String[] args) {
        int Pid,RW,Addr,Timestamp;
        Pid=1;RW=0;Addr=3;Timestamp=9;
        System.out.println("!PROG: assert ...";
        Pid:"+Pid+";RW:"+RW+";Addr:"+Addr+";Val:"+Val+";Timestamp:"+Timestamp+";!");
    }
}
```
Figure 4.9 illustrates the compilation of the external Java program in Listing 4.3. The compilation occurs within AnaTempura.

![Figure 4.9: Generating Extended Assertion Points within External Java Program](image)

The new functions in lines 4 to 9 Listing 4.4 are extensions of the previous ones in Listing 4.2. The function in line 4 and 5 always reserve the parameters $X[0]$ and $X[1]$ to $\text{Pid}_{\text{var}}$ and $\text{Pid}_{\text{val}}$ respectively. The rest of the functions in line 6 and 7 have new parameters $a$ and $b$ to enable their functions to assign corresponding variables to their values dynamically. The time stamp in microseconds is introduced in line 9.

Listing 4.4: Collecting Assertion Data within Tempura Program

```java
load "../library/conversion".
load "../library/exprog".
/* java ExtendedAssertionPoints 0 */
define apidvar(X) = {X[0]}.
define apidval(X) = {X[1]}.
define avar1(X,a) = {X[a]}.
define avall(X,b) = {X[b]}.
define atime1(X,c) = {strint(X[c])}.
define atime_micro1(X,d) = {strint(X[d])}.
set print_states = true.
define get_var(Variable0,Value0,Variable1,Value1,Variable2,Value2,Timestamp) = {
  exists T : {
    get2(T) and
    Variable0 = apidvar(T) and Value0 = strint(apidval(T)) and
    Variable1 = avar1(T, a) and Value1 = strint(avar1(T, a)) and
    Variable2 = avall(T, b) and Value2 = strint(avall(T, b)) and
    get2(T) and
    Timestamp = atime1(T, c) and
    Timestamp = atime_micro1(T, d)
  }
}
```
CHAPTER 4. DESIGN AND IMPLEMENTATION OF A PARALLEL RUNTIME
VERIFICATION FRAMEWORK (PRVF)

Variable1 = avar1(T,2) and Value1 = strint(aval1(T,3)) and
Variable2 = avar1(T,4) and Value2 = strint(aval1(T,5)) and
Timestamp = atime_micro1(T,6) and
format("Assertion data <%s, %d, %s, %d, %s, %d> are received\n",
Variable0,Value0,Variable1,Value1,Variable2,Value2,Timestamp)
}

/* run */ define Test() = {
exists Variable0,Value0,Variable1,Value1,Variable2,Value2,Timestamp: {
get_var(Variable0,Value0,Variable1,Value1,Variable2,Value2,Timestamp) and len(0)
}
}. 

Figure 4.10 illustrates the collection process of a generated assertion data sent from the external Java program. The assertion data get assigned to their functions accordingly as described earlier in this section.

Figure 4.10: COLLECTING EXTENDED ASSERTION POINTS TEMPURA PROGRAM

• The Tempura Interpreter can now be several instances to handle concurrency. This new capability allows us to generate as many Tempura Interpreters as needed. To run the Tempura Interpreter externally, a certain command has to be annotated within the main Tempura file that is intended to monitor other Tempura programs. For instance, Listing 4.5 is a global Tempura program which is intended to monitor two local Tempura programs illustrated in Listings 4.6 and 4.7. The global program in Listing 4.5 starts another AnaTempura system
via these commands in lines 3 and 4:

```plaintext
/* anatempura 0 */
/* anatempura 1 */
```

A description of this new capability will be given where relevant in this section. In order to start several Tempura Interpreters instead, these commands are replaced by the new commands at the top of the Global Tempura program in line 3 and 4:

```plaintext
/* prog tempura_macosx 0 */
/* prog tempura_macosx 1 */
```

These new commands start C-Tempura Interpreters as external programs within the main monitoring system, AnaTempura, in order to monitor local programs behaviour via generating assertion points and sending the assertion data to the global Tempura program.

- The ability of the Monitor to monitor global and local properties via collecting the assertion data that are sent from local programs. For instance, local programs in Listings 4.6 and 4.7 send their assertion data to global program in Listing 4.5. The output as illustrated in Figure 4.11 where the monitor at the top of the figure and `local0` and `local1` are at the middle and the bottom respectively.

Listing 4.5: Collecting Assertion Data within Tempura Program

```plaintext
load "conversion".
load "exprog".
/* anatempura 0 */
```
CHAPTER 4. DESIGN AND IMPLEMENTATION OF A PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

/* anatempura 1 */

define apidvar(X) = {X[0]}.
define apidval(X) = {X[1]}.
define avar1(X,a) = {X[a]}.
define aval1(X,b) = {X[b]}.
define atime1(X,c) = {strint(X[c])}.
define atime_micro1(X,d) = {strint(X[d])}.
set print_states = true.
define get_var() = {
exists T : {
    get2(T) and
    format("Global is Receiving Assertion Data: %s=%20d from %s %d\n",
    avar1(T,2),strint(aval1(T,3)),apidvar(T),strint(apidval(T))) and empty
}
}. /* run */ define test() = {
exists v : {
    {prog_send1(0,"load 'Local0'.") and
    prog_send1(1,"load 'Local1'.");skip;
    {prog_send1(0,"run test_local0().") and
    prog_send1(1,"run test_local1().");skip;
    for v<2 do {
        {get_var();skip}
    }
    {prog_send1(0,"exit.")) and prog_send1(1,"exit."))
    }
}.  

* The possibility to monitor a Tempura Interpreter (or another AnaTempura system) so a hierarchy of monitors exist. A description of the process of monitoring another Tempura Interpreter is given earlier. The process of monitoring another monitoring system AnaTempura can be done via the annotation of a certain command:

/* anatempura 0 */
/* anatempura 1 */
For instance, the global program run within Listing 4.5 has this command in lines 3 and 4. This command runs AnaTempura and this task is assigned to process 0 such as in line 3, and process 1 in line 4. The global Tempura program runs two local Tempura programs independently in parallel to monitoring their behaviours in order to make a judgement according to a set of properties. The functions within a global program which are intended to load the local programs are:

\[
\text{prog\_send}(\text{Pid}, "load 'Program'".)
\]

for instance,

\[
\text{prog\_send1}(0, "load 'Local0'") \text{ and } \text{prog\_send1}(1, "load 'Local1'").
\]

The first parameter is a process \textbf{Pid} which is intended to load local program ‘local0’. The same steps are applied to the rest of local programs when they ever exist, while the functions which are intended to run functions within locals programs as follows:

\[
\text{prog\_send}(\text{Pid}, "run Function".)
\]

for instance,

\[
\text{prog\_send1}(0, "run test\_local0()") \text{ and } \text{prog\_send1}(1, "run test\_local1()").
\]

To run a certain function externally, the same value for the process \textbf{Pid} which has been used to load this function. The difference here is the use of “run” keyword instead of “load”.

85
The Listing 4.6 illustrates local Tempura program. This program is loaded within the global Tempura program as explained above, and the function as well is run externally within the global Tempura program.

Listing 4.6: Generating Assertion Data within local0 Tempura Program

```
load "conversion".
load "exprog".
set print_states = false.
define assert() = {
    exists Local,Data : {
        Local=0 and
        Data=Random and
        format("\n") and
        format("Local %d is Sending %d to Global\n",Local,Data) and
        format("!PROG: assert Local:%d:X:%d:!\n",Local, Data)
    }
}
/* run */ define test_local0() = {
    skip and assert()
}.
```

The local Tempura programs in Listings 4.6 and 4.7 are alike except in variables Local and Data. The variable Local’s value is 0 in Listing 4.6 while it is 1 in Listing 4.7. The variable Data is generated randomly by assigning the random operator, Random, as a value to it.

Listing 4.7: Generating Assertion Data within local1 Tempura Program

```
load "conversion".
load "exprog".
set print_states = false.
define assert() = {
    exists Local,Data : {
        Local=1 and
```

86
The monitor then displays the assertion data which are generated by local Tempura programs and collected via a global Tempura program as illustrated in Figure 4.11.

Figure 4.11: GLOBAL COLLECTS ASSERTION POINTS FROM LOCALS TEMPURA PROGRAM

- The integration between AnaTempura and Java Remote Method Invocation (RMI) Framework. AnaTempura allows plug-ins, as external systems, systems which use Java RMI to
start a server implementation in order to serve clients systems run in parallel using multi-
threads programming in Java language. The compilation of Java RMI programs is unlike
other Java programs; it has different steps. The first step is to start a server and then run
the independent clients upon a running server. These steps are now embedded within the
Tempura Interpreter. In Listing 4.8 lines 3-6 are the commands which run Java RMI pro-
grams:

```java
/*RMIREGISTRY 0*/
/*RMISERVER . RmiServerIntf RmiServer 1*/
/*RMICLIENT . RmiServerIntf RmiClient1 2*/
/*RMICLIENT . RmiServerIntf RmiClient2 3*/
```

The creation of RMI registry is assigned to process 0. The compilations of java programs
RmiServer, RmiClient1, RmiClient2 are assigned to processes 1, 2, 3 respectively.
CHAPTER 4. DESIGN AND IMPLEMENTATION OF A PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

18 Data=strint(avall(T,3)) and
19 Timestamp =atime_microl(T,4) and
20 format("Server is Receiving Assertion Data: X=%ld from Client %d at timestamp ... %s\n", 
21 Data,Client,Timestamp) and empty
22 }
23 }
24 /* run */ define test() = {
25 exists v :
26 for v<2 do {get_var();skip}
27 }
28 }

Figure 4.12: IMPLEMENTATION JAVA RMI USING AnaTEMPURA
Listing 4.8 runs Java programs associated with it in lines 4, 5, 6 as seen in Figure 4.12. The clients Java programs have assertion points and once these programs are run via AnaTempura, they send their assertion data to their corresponding Tempura programs to receive the assertion data accordingly and then forward these assertion data as assertion points to Tempura program in Listing 4.8. Then, the assertion data is received and displayed as seen in Figure 4.12. The clients Java programs assert a random data, time stamp in microseconds using a format of HH-mm-ss-SSS where HH stands for Hours, mm stands for minutes, ss stands for seconds, and SSS stands for milliseconds. The source code of these Java programs and their relevant Tempura programs can be found in Appendix D.

- The capability to implement parallel systems designed using multi-core processor architectures. The case study, Cache Controller, is a demonstration of this capability in the next chapter, Chapter 5.

4.4 Benchmarking Applications

In this section, some parallel/concurrent applications which can be applied using the proposed implemented framework are explored. Producer-Consumer and Dining Philosophers Problem are two common applications that demonstrate parallel/concurrent executions.

4.4.1 Producer-Consumer

The Producer and Consumer are two separate, concurrent programs which run in parallel and share the same data. The access to shared data must be synchronised to deliver a consistent model. A producer puts (produces) a stream of data into a buffer, while a consumer gets (consumes) these produced data within a buffer as Figure 4.13 illustrates.

![Figure 4.13: Producer-Consumer](image-url)
A buffer in this example of Producer-Consumer can hold up to four elements. When the size of the buffer is full, it can not accept new produced elements by the producer. In such cases, the producer waits until the buffer empties a space for a new element. The implementation of Producer-Consumer using the proposed model in the runtime verifier AnaTempura is illustrated in Figure 4.14.

The implementation shows the assertion data being asserted within Java external programs that are intended to run a Producer-Consumer system in order to analyse its behaviour in order to check desired correctness properties of such programs. As seen in Figure 4.14, the assertion data are displayed in the monitor’s window (left side) and in the simulation window as well (right side). Based on these data, a complete check of correctness properties can be achieved.
4.4.2 Dining Philosophers Problem

The Dining Philosophers Problem is a classical example of parallel/concurrent programs. Five philosophers are sitting around a circular table. The five philosophers are either thinking or eating spaghetti. Eating spaghetti needs two chopsticks, but unfortunately only five chopsticks are available. Each philosopher has two chopsticks; they are to his/her immediate right and left. When a philosopher uses two chopsticks, it means his/her immediate neighbours can not eat because the chopsticks they need to pick up are taken and unavailable. The Dining Philosophers Problem demonstrates how to provide a synchronisation mechanism that ensures correctness properties in such cases. Figure 4.15 illustrates an implementation of this problem which runs in AnaTempura.

Figure 4.15: DEMO OF DINING PHILOSOPHERS PROBLEM

Figure 4.15 illustrates that there are five parallel/concurrent programs running simultaneously. Each program represents a philosopher that is assigned to $P_{id}$’s (0 to 4). A philosopher’s
actions are thinking (default initial action), eating, picking up a (left/right) chopstick, and putting down a (left/right) chopstick. Chopsticks are numbered as well to identify them; they are numbered from 0 to 4 as Figure 4.15 illustrates.

Five external programs model these five philosophers and their actions. The most critical part is the use of chopsticks because they are shared. Two neighbour philosophers are not allowed to use the chopstick they share, e.g., Philosopher 0 and Philosopher 1 share Chopstick 0 and so on.

The proposed model handles this problem perfectly, and it allows the five parallel programs to run and generate assertion data and displays these data in a table format within the monitor and in graphics using the simulation window. These features allow modelling such applications in order to validate their behaviour against correctness properties.

4.5 Summary

In this chapter, Parallel Runtime Verification Framework (PRVF) is designed and implemented. A review of the current status of the runtime verifier AnaTempura is given, and the drawbacks in this model with regards to handling parallel systems are addressed. Then, a mechanism of how to handle parallel systems natively using the proposed model is illustrated. Applications such as Producer-Consumer and Dining Philosophers Problem are implemented using this model.
Chapter 5

Case Study: Cache Controller

Objectives:

• To present a Case Study: the Cache Controller

• To review Cache Coherence and implement MSI Protocol

• To produce a Formal Specification in Interval Temporal Logic (ITL)

• To deliver a Runtime Verification using Tempura/AntaTempura
5.1 Cache Memory Controller: A Case Study

In this chapter, a case study of Private L2 Cache Memory Controller that will illustrate our compositional model is introduced. A comprehensive description of the operations and requests of Private L2 Cache Memory, Processor, Main Memory, and MSI Protocol is given.

5.2 The Basics of Cache Memory

According to Webster’s New World Dictionary of the American Language (Third College Edition 1988) a cache is “a safe place for hiding or storing things.”; to exemplify, consider a university library as the main memory, and the desk as the cache, the books are the things that must be found [213]. Ever since the first appearance of the caches in research computers in 1960s and then in computers production, they have been included in every built computer today [213].

Assigning the cache location based on the address of the word in the main memory is the simplest method to assign a location in the cache. The process of mapping in a direct way of each memory location to exactly one location in the cache is called direct-mapped cache. This mapping can be easily done by applying the modulo mathematical operation which always gives the remainder of the division operation of two operands. For instance, to find a block in direct-mapped cache, the following equation is used:

\[
\text{Index} = X \mod Y
\]  

(5.1)

where \(X\) is a decimal address, and \(Y\) is the number of blocks or entries in the cache, in the case it is a power of 2. To compute the length of the index, the low order is used. In Equation 5.2, \(S\) is a cache size in blocks and can be the exponential multiples of the base 2, such as 2, 4, 8, 16, 32, 64, 128 etc.

\[
\log_2(S)
\]  

(5.2)
Assuming that there are eight bits length for the requested address, the length of bits of cache’s index can be found by computing the following:

\[ \log_2(8) = 3 \text{ bits} \]

This means that there are eight blocks \(2^3\) in the cache which are \(000, 001, 010, 011, 100, 101, 110, 111\).

Suppose there are 10 as a decimal address requested by a processor, and the length of the block address is 8 bits, the requested address has to go through the following:

1. Convert the requested address into binary: \(10_{10} = 01010_2\)

2. Determine the length of bits used for cache index: \(\log_2(8) = 3\) bits

3. Modulo used to determine the cache index that will match this address:
   \[ 10_{10} \mod 8_{10} = 2_{10} \text{ or in binary format } 010_2 \]

Therefore, the requested address \(10_{10}\) goes to index \(010_2\) and continues the computation. But, this index could be shared by other requested addresses such as \(18_{10}, 26_{10}, 34_{10}\) or any decimal number having \(2_{10}\), or alternatively \(010_2\), as a resultant of the modulo operation. To solve this conflict the tag field is introduced. Tags contain the upper portion of the address to distinguish this requested address from other addresses which have the same index block. For instance, consider previous example:

| Address 10\(_{10}\) : 10\(_{10}\) mod 8\(_{10}\) | 2\(_{10}\) or 010\(_2\) |
| Address 18\(_{10}\) : 18\(_{10}\) mod 8\(_{10}\) | 2\(_{10}\) or 010\(_2\) |

Both addresses have the same index. Therefore, if the two upper portions are set of the binary
address as a tag field, then there will be different tags which are:

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>10_{10}</td>
<td>01_2</td>
</tr>
<tr>
<td>18_{10}</td>
<td>10_2</td>
</tr>
</tbody>
</table>

Alternatively, the tag field can be determined using the division operation of the requested address over the the length of the cache index as Equation 5.3 illustrates:

\[
Tag = \frac{Addr}{S}
\]  

(5.3)

where \( Addr \) is the requested address and \( S \) is the size of the cache. For instance, in case the size of the index is 8 and to determine the tag field of addresses 10, 18, Equation 5.3 is used as follows:

\[
\begin{align*}
10 \div 8 &= 1_{10} \text{ or } 01_2 \\
18 \div 8 &= 2_{10} \text{ or } 10_2
\end{align*}
\]

Therefore, the addresses from 0 to 7 have the tag 0, the addresses from 8 to 15 have the tag 1, and the addresses from 16 to 23 have the tag 2 and so on.

5.2.1 Description

The multi-core processor architecture has at least two independent cores, each core has its L1 cache, and they share L2 cache as illustrated in figure 5.1. Some architectures have different designs such as shared L2 cache; the dedicated or private L2 cache design is adopted to demonstrate the proposed approach. The main memory is connected to the L2 cache memory using a bus. The bus is a broadcast medium that transients the addresses and data requested by the processors between the caches or between the cache and main memory.
A core or a processor requests either a read or a write operation. When a processor requests to read an address from the cache, the cache checks its index; if it is found, then the cache fetches the address to the processor. This case is called Read Hit. If the cache does not find the requested address within its index, the request gets transferred to the main memory and the main memory fetches the requested address to the processor, and it keeps this address in the cache for further requests by the processor. This case is called Read Miss.

When the request is write, it becomes more complicated. There are two types of write operation which are are write-through and write-back. In write-through, the write operation updates both the cache and the main memory simultaneously, so the cache and the main memory are always consistent. In write-back when a write occurs, it updates the cache only, then writes the modified block of the cache to the main memory when the block is replaced [213].

In case of write-through, the processor requests to write data to a block in the cache; if the block is found, the data is then written to the block in the cache and at the same moment the main memory gets updated. This case is called Write Hit. If the requested cache block is not found then the request gets transferred to the main memory and performs the write operation upon the
requested address. Then the main memory keeps a copy of this updated block in the cache for further requests. This case is called Write Miss.

In case of write-back, a processor requests to write data to a cache block; if a cache block is found, then the write operation occurs without updating the main memory, which means that the cache block and the main memory are inconsistent. This case is called Write Hit. This scheme of writing improves the performance of the processor as the processor does not need to wait until the main memory becomes consistent with the cache. Instead, the processor continues performing other tasks. But whenever that cache block gets replaced by another request, the modified block gets written to the main memory. If the requested cache block is not found in the cache, then the main memory fetches the data of the requested address to the correspondent cache block. This case is called Write Miss[213]. Tables 5.1 to 5.7 illustrate the read-write/miss-hit of the addresses requested by a processor. Table 5.1 illustrates 9 requests by a processor of addresses Memory[22], Memory[26], Memory[16], Memory[3], and Memory[18]. Some of these addresses are requested twice which causes the occurrence of hits within the cache. For instance, at state 0, a processor requests address Memory[22] and because this address is not present within the cache, the request is transferred to the main memory to deliver it to the requester. The main memory of address 22, Memory[22], gets copied into the cache accordingly, and the requested data of this memory address is provided to the processor to continue the computation. The same

<table>
<thead>
<tr>
<th>State</th>
<th>Decimal Address</th>
<th>Binary Address</th>
<th>Hit-Miss</th>
<th>Assigned Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22_{10}</td>
<td>10110_{2}</td>
<td>Miss</td>
<td>(10110_{2} mod 8) = 110_{2}</td>
</tr>
<tr>
<td>1</td>
<td>26_{10}</td>
<td>11010_{2}</td>
<td>Miss</td>
<td>(11010_{2} mod 8) = 010_{2}</td>
</tr>
<tr>
<td>2</td>
<td>22_{10}</td>
<td>10110_{2}</td>
<td>Hit</td>
<td>(10110_{2} mod 8) = 110_{2}</td>
</tr>
<tr>
<td>3</td>
<td>26_{10}</td>
<td>11010_{2}</td>
<td>Hit</td>
<td>(11010_{2} mod 8) = 010_{2}</td>
</tr>
<tr>
<td>4</td>
<td>16_{10}</td>
<td>10000_{2}</td>
<td>Miss</td>
<td>(10000_{2} mod 8) = 000_{2}</td>
</tr>
<tr>
<td>5</td>
<td>3_{10}</td>
<td>00011_{2}</td>
<td>Miss</td>
<td>(00011_{2} mod 8) = 011_{2}</td>
</tr>
<tr>
<td>6</td>
<td>16_{10}</td>
<td>10000_{2}</td>
<td>Hit</td>
<td>(10000_{2} mod 8) = 000_{2}</td>
</tr>
<tr>
<td>7</td>
<td>18_{10}</td>
<td>10010_{2}</td>
<td>Miss</td>
<td>(10010_{2} mod 8) = 010_{2}</td>
</tr>
<tr>
<td>8</td>
<td>16_{10}</td>
<td>10000_{2}</td>
<td>Hit</td>
<td>(10000_{2} mod 8) = 000_{2}</td>
</tr>
</tbody>
</table>
steps are taken at state 1 of address 26. The interesting part is that when one of the previous addresses gets requested again by a processor, it means that the requested address is already now in the cache after the fetch operation is performed by the main memory in the previous state. The request of the requested address gets hit as illustrated in table 5.1 at state 2 and 3 of addresses Memory[22] and Memory[26]. The same policy is applied on the remaining requests.

Tables 5.2 to 5.7 show the described policy of read-write/miss-hit step by step. The tables are designed according to the cache main components. The cache memory has Index, Valid, Tag, and Data fields. The index is a unique place to store the requested addresses with their data accordingly as illustrated in Equation 5.1. The tag determination is described in Equation 5.3. The valid bit is an indication of whether the cache block is empty or not. For instance, it might have 0 or N to indicate that the cache block is not valid because it is empty, whereas the values 1 or Y indicate that the cache block is valid.

<table>
<thead>
<tr>
<th>Table 5.2: Empty 8-Blocks Cache</th>
<th>Table 5.3: Miss of Address [10110&lt;sub&gt;2&lt;/sub&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>V</td>
</tr>
<tr>
<td>000</td>
<td>N</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
</tr>
<tr>
<td>010</td>
<td>N</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
</tr>
<tr>
<td>101</td>
<td>N</td>
</tr>
<tr>
<td>110</td>
<td>N</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.4: Miss of Address [11010&lt;sub&gt;2&lt;/sub&gt;]</th>
<th>Table 5.5: Miss of Address [10000&lt;sub&gt;2&lt;/sub&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>V</td>
</tr>
<tr>
<td>000</td>
<td>N</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
</tr>
<tr>
<td>101</td>
<td>N</td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
</tr>
<tr>
<td>Index</td>
<td>V</td>
</tr>
<tr>
<td>000</td>
<td>Y</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
</tr>
<tr>
<td>101</td>
<td>N</td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
</tr>
</tbody>
</table>
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

Table 5.6: Miss of Address [00011₂]

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10₂</td>
<td>Memory [10000₂]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11₂</td>
<td>Memory [1101₀₂]</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00₂</td>
<td>Memory [0001₁₂]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10₂</td>
<td>Memory [1011₀₂]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Miss of Address [10010₂]

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10₂</td>
<td>Memory [10000₂]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>10₂</td>
<td>Memory [1001₀₂]</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00₂</td>
<td>Memory [0001₁₂]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10₂</td>
<td>Memory [1011₀₂]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2.2 MSI Protocol

To maintain cache coherence for multi-core architecture, the cache coherence protocols are implemented. Snooping protocol is the most popular cache coherence protocol. The key to implement these protocols is the track of the states of the caches’s blocks. A cache block has different states, when the the block is shared by more than one processor, it is called the Shared state or is simply represented as S. The Modified state or M state is the state when the block is modified in the cache, and this block is not consistent with the main memory. When a cache block gets modified by a processor, any other processors with copies of this cache block has to invalidate their copies; it is represented as I state. These three states together form a protocol called MSI Protocol. There are other protocols with extended states such as MESI with the Exclusive state E, and another protocol is called MOESI with another state called Owned or O [213]. However, in this research, the simplest protocol which is MSI Protocol is adopted to demonstrate the proposed approach as these three states perfectly serve the case study.

This protocol is proposed to maintain coherence of the cache memory of one processor with another cache memory of a different processor. The Modified state occurs when a cache block is inconsistent with its correspondent in the main memory. The Shared state occurs when a cache block is consistent with another processor’s same cache block or with its correspondent in the main memory or both. The Invalid state occurs when a cache block is not present in the cache.
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

or updated in another cache block of another processor. Any two or more processors which have
their private cache memory must meet the criteria of MSI Protocol in Table 6.1.

Table 5.8: MSI Protocol

<table>
<thead>
<tr>
<th></th>
<th>Modified</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Shared</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Invalid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The check mark means that any two or more cache blocks of different processors can have
these states at the same time, while the cross mark means the occurrence of these states is not
allowed at the same time.

5.2.3 Formal Description of Cache Controller

The basic operations and properties of Processor, Level 2 of Cache Memory (L2 Cache), Mem-
ory, and MSI Protocol are summarised as follows:

- Operations of the processor:
  1. Read from Address A (0 indicates Read operation)
  2. Write Data B to Address A (1 indicates Write operation)

- Status of the processor’s request:
  1. Hit
  2. Miss

- Status of L2 Cache Index:
  1. Valid (0 indicates Invalid, 1 indicates Valid), where Valid means that the cache block
     is not empty and it has a datum in it.
2. Dirty (0 indicates Not Dirty, 1 indicates Dirty), where Dirty means that the cache block is not consistent with the main memory.

- MSI Status of L2 Cache Index:
  1. Modified (If the index is inconsistent with its correspondent in the Main Memory.)
  2. Shared (If the index is consistent with another processor’s cache block or the main memory or both.)
  3. Invalid (If the index is not present in the cache or updated in another cache.)

5.2.4 Compositional Modelling

In this section, a compositional modelling of the behaviour of the components of Cache Memory Controller System using Interval Temporal Logic (ITL) is given as follows:

1. Processor\[i\] (0 ≤ i < nprocessors), where nprocessors = 3

2. L2CacheTag\[i][j] (0 ≤ i < nprocessors), (0 ≤ j < ncachelocations), where ncachelocations = 8

3. L2CacheState\[i][j] (0 ≤ i < nprocessors), (0 ≤ j < ncachelocations)

4. Valid\[i][j] (0 ≤ i < nprocessors), (0 ≤ j < ncachelocations)

5. Dirty\[i][j] (0 ≤ i < nprocessors), (0 ≤ j < ncachelocations)

6. L2CacheMemory\[i][j] (0 ≤ i < nprocessors), (0 ≤ j < ncachelocations)

7. MainMemory\[j] (0 ≤ j < nmemorylocations), where nmemorylocations = 16

A formal description in Interval Temporal Logic (ITL) [54] of the Cache Controller system is given. The possible transitions of the system are as follows:
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

The Processor X request: Let Processor[X] be a state variable representing the state of Processor X with all possible values with regards to Read-Write/Hit-Miss. The specification expressed in Interval Temporal Logic (ITL) is a formal description of Processor X Requests. The full specification of this behaviour is written in Tempura code in Appendix B. Tempura is an executable subset of Interval Temporal Logic (ITL). Refer to table 5.9, for more details. I refer the reader to [182]. The following are variables declarations and their descriptions:

\( X = \text{Random mod } 3 : \) the case study has three processors

\( Y = (\text{Random} + 1) \mod 3 \)

\( Z = (\text{Random} + 2) \mod 3 \)

\( RW = \text{Random mod } 2 : \) if \( RW = 0 \) it is Read, if \( RW = 1 \) it is Write

\( \text{Addr} = \text{Random mod } 16 : \) Random generation of addresses between 0 and 15

\( \text{Tag} = \text{Addr} \div 8 : \) Tag used to distinguish the addresses which share the cache’s index

\( \text{Data} = \text{Random mod } 30 : \) Random generation of the data between 0 and 29

\( \text{Indexc} = \text{Addr} \mod 8 : \) The size of the cache is 8 blocks

\( \text{Indexm} = \text{Addr} \mod 16 : \) The size of the memory is 16 blocks

\( \text{Indexm}’ = \text{Addr} \mod 16 : \) where \( \text{Indexm} \neq \text{Indexm}’ \)

\( \text{InitialValuec} = -8 : \) The initial value for cache blocks is \(-8\)

\( \text{InitialValuem} = -16 : \) The initial value for memory is \(-16\)

\( \text{tagx} = \text{L2CacheTag[X][Indexc]} : \) Tag of cache block indexc of Processor X

\( \text{tagy} = \text{L2CacheTag[Y][Indexc]} : \) Tag of cache block indexc of Processor Y

\( \text{tagz} = \text{L2CacheTag[Z][Indexc]} : \) Tag of cache block indexc of Processor Z

\( \text{ntagx} = \circ (\text{L2CacheTag[X][Indexc]}) : \) Next state Tag of cache block indexc of Processor X

\( \text{ntagy} = \circ (\text{L2CacheTag[Y][Indexc]}) : \) Next state Tag of cache block indexc of Processor Y

\( \text{ntagz} = \circ (\text{L2CacheTag[Z][Indexc]}) : \) Next state Tag of cache block indexc of Processor Z

\( \text{csx} = \text{L2CacheState[X][Indexc]} : \) State of cache block indexc of Processor X
The main operations in the cache controller system are read and write. A formal expression of read and write operations in ITL is considered later in this section. The rest of the operations can be derived and expressed in ITL by referring to Table 5.9. The read operation occurs when the marker RW’s value is 0. There are three processors which are X, Y and Z, where they individually check values of the relevant variables in order to deliver coherence cache states and consistent memory. The following specifications are modelling the read operation in ITL:
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

Processor_Request(X, RW, Addr, Data) \equiv (  

Skip \land 

if RW = 0 then ( 

if Tag = Tagx \land 

Statex = shared \lor 

Statex = modified then ( 

stringx := Read Hit \land 

stable(Mem[Indxm]) \land 

stable(L2Cache[X][Indxc]) \land 

stable(Valid[X][Indxc]) \land 

if Statex = modified then ( 

Dirty[X][Indxc] := 1 \land 

Statey := invalid \land 

Statez := invalid 

) else ( 

stable(Dirty[X][Indxc]) \land 

stable(State[Y][Indxc]) \land 

stable(State[Z][Indxc]) 

))) 

if Tag = Tagy \land 

Statey = shared \lor 

Statey = modified then (\ldots) 

if Tag = Tagz \land 

Statez = shared \lor 

Statez = modified then (\ldots)
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

The case for the write operation is encountered when the marker RW’s value is 1. The following specifications are modelling the write operation in ITL:

\[
\text{Processor\_Request}(X, RW, Addr, Data) \equiv ( \\
\text{Skip} \land \\
\text{.} \\
\text{.} \\
\text{else if } RW = 1 \text{ then (} \\
\text{if } Tag = Tagx \text{ then (} \\
\text{stringx} := \text{Write Hit} \land \\
\text{L2Cache}[X][Indxc] := Data \land \\
\text{stable(Mem[Indxm])} \land \\
\text{stable(Valid[X][Indxc])} \land \\
\text{if L2Cache}[X][Indxc] \neq Mem[Indxm] \text{ then (} \\
\text{Dirty[X][Indxc]} := 1 \land \\
\text{Statx} := \text{modified} \land \text{Staty} := \text{invalid} \land \text{Statz} := \text{invalid} \\
\text{)} \text{ else (} \\
\text{stable(Dirty[X][Indxc])} \land \text{stable(State[X][Indxc])} \land \\
\text{stable(State[Y][Indxc])} \land \text{stable(State[Z][Indxc])} \\
\text{)} \text{ else (} \\
\text{stringx} := \text{Write Miss} \land \\
\text{Mem[Indxm]} := Data \land \\
\text{if Dirty[X][Indxc]} = 1 \text{ then (} \\
\text{Mem[Indxm']} := L2Cache[X][Indxc] \land \\
\text{○(L2Cache[X][Indxc] := Mem[Indxm])} \land \\
\text{○(Dirty[X][Indxc]} := 0) \land \text{○(Statex} := \text{shared}) \land \\
\text{○(Staty} := \text{invalid}) \land \text{○(Statz} := \text{invalid}) \land 
\]
 CHAPTER 5. CASE STUDY: CACHE CONTROLLER

50  \((\text{stable}(\text{Valid}[X][\text{Indxc}]))\)
51  \)
52  \(\text{L2Cache}[X][\text{Indxc}] := \text{Mem}[\text{Indxm}] \land\)
53  \(\text{Valid}[X][\text{Indxc}] := 1\)
54  )))
55  \(\text{if} \ Tag = \text{Tagy} \text{ then } (\cdots)\)
56  .
57  .
58  \(\text{if} \ Tag = \text{Tagz} \text{ then } (\cdots)\)
59  .
60  .
61  )

For a complete ITL modelling for the cache controller case study, the Tempura code is listed in Appendix B where Table 5.9 can be used as a conversion from Tempura to ITL syntax.

Table 5.9: TEMPURA SYNTAX VERSUS ITL SYNTAX

<table>
<thead>
<tr>
<th>ITL</th>
<th>Tempura</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1 \land f_2)</td>
<td>(f_1 \text{ and } f_2)</td>
</tr>
<tr>
<td>(A := \text{exp})</td>
<td>(A := \text{exp})</td>
</tr>
<tr>
<td>(\Diamond)</td>
<td>sometimes</td>
</tr>
<tr>
<td>(\Box)</td>
<td>always</td>
</tr>
<tr>
<td>(\bigcirc)</td>
<td>next</td>
</tr>
<tr>
<td>if (b) then (f_1) else (f_2)</td>
<td>if (b) then (f_1) else (f_2)</td>
</tr>
<tr>
<td>while (b) do (f)</td>
<td>while (b) do (f)</td>
</tr>
<tr>
<td>Repeat (b) Until (f)</td>
<td>Repeat (b) Until (f)</td>
</tr>
<tr>
<td>procedures</td>
<td>define (p(e_1,\ldots,e_n) = f)</td>
</tr>
<tr>
<td>functions</td>
<td>define (g(e_1,\ldots,e_n) = e)</td>
</tr>
</tbody>
</table>
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

**Processor X Writes to Cache:** When a requested address is found in a cache, then a write operation occurs in the cache which belongs to Processor X. VBit changes its value to 1 to indicate that this cache block is valid. The tag of this cache block changes its value to the tag determined by applying Equation 5.3 on Processor X and the requested address.

\[
\text{write_to_cache}(L2CacheMemory, L2CacheTag, Vbit, X, M, V, tag, j) \equiv ( \\
\text{skip} \land \\
(\forall i < nprocessors \bullet \\
(\forall j < ncachelocations \bullet \\
\text{if } i = X \land j = M \text{ then} ( \\
\text{if } Vbit[i][j] = 1 \text{ then (stable}(Vbit[i][j]) \\
\text{) else } (Vbit[i][j] := 1) \land \\
L2CacheTag[i][j] := tag \land \\
L2CacheMemory[i][j] := V \\
\text{) else (}
\text{stable}(Vbit[i][j]) \land \\
\text{stable}(L2CacheTag[i][j]) \land \\
\text{stable}(L2CacheMemory[i][j])
\text{) })
\]

**Processor X Writes to Memory:** When Processor X requests to write to the cache block and this cache block is already occupied by another address, the data of this address gets written to a memory and a new requested address writes its new data to this cache block and sets Dirty Bit to 1 to indicate that this cache block and its correspondent in the memory are inconsistent.
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

\textbf{write}\_\textit{to}\_\textit{memory}(\textit{MainMemory}, X, M, V, Tick) \equiv ( \\
\textbf{skip} \land \\
(\forall j < n\textit{memorylocations} \bullet ( \\
\textbf{if} \ j = M \ \textbf{then} \ (\textit{MainMemory}[j] := V) \\
\textbf{else} ( \\
\textbf{stable} (\textit{MainMemory}[j]) \\
) \\
) \\
) \\
)

\textbf{Memory is unchanged:} At every state, the memory either gets changed or unchanged. The cases when a memory is unchanged, is the case where a write-back occurs.

\textbf{memory}\_\textit{unchanged}(\textit{MainMemory}) \equiv ( \\
\textbf{skip} \land \\
(\forall j < n\textit{memorylocations} \bullet ( \\
\textbf{stable} (\textit{MainMemory}[j]) \\
) \\
) \\
)

\textbf{Cache is unchanged:} When a read hit occurs, then a cache stays unchanged. Otherwise, a cache gets changed.

\textbf{cache}\_\textit{unchanged}(L2\textit{CacheMemory}, L2\textit{CacheTag}, V\textit{bit}, x) \equiv ( \\
\textbf{skip} \land \\
(\forall j < n\textit{cachelocations} \bullet ( \\
) \\
) \\
)
stable(Vbit[x][j]) ∧
stable(L2CacheTag[x][j]) ∧
stable(L2CacheMemory[x][j])
)
)
)

**Update MSI States:** The states of MSI Protocol has been discussed in section 5.2.2, and the criteria that manages these states is illustrated in table 6.1.

update_msi(i, B, L2CacheState, v) ≜ ( ∧
(∀ j < ncachelocations • if j = B then (L2CacheState[i][j] = v)
else ( stable(L2CacheState[i][j])
)
)
)
)

**5.3 Analysis and Discussion**

In this section, data analysis of the collected data after the execution of Parallel Runtime Verification Framework (PRVF) on the cache controller case study is given. Figure 5.2 demonstrates the final execution of the cache controller. In this case study, an assumption has been made in which there are three independent processors running in parallel in order to demonstrate the cache controller system’s behaviour in order to check global correctness properties of such a system.
Figure 5.2: CACHE CONTROLLER
5.3.1 Global Program : Cache Controller

Figures 5.3 and 5.4 illustrate the execution of the first state, state 0, of the cache controller. For all states execution (state 0 to state 9) of the case study, see appendix A. Figure 5.3 is the output of the run of a Tempura code of the cache controller at state 0, while Figure 5.4 is a graphical simulation of the cache controller written in Tcl/Tk language [288, 179, 155] of the same state number.

The output shows the details of the request which has been made by a random processor. The request is either a read or write request. Every request has the ID of a requester processor X, the read-write indicator RW (0 for read, 1 for write), the requested address in the memory Addr, and the data Data which is written either to a cache memory or a main memory, or both.

In case the read-write indicator is 0, which means read operation, the data field is used to store the requested value of the requested address either from the cache memory in case the request gets hit or from the main memory in case the request gets missed.

Therefore, at every state this information has to be shown in details. This information includes State number, Processor ID, Address in the memory, Read-Write indicator, and Data. Based on these data, expanded information is given within the table in figure 5.3. This information is illustrated in Table 5.10.

5.3.1.1 Raw Data Description

The first column Table 5.10 is the state number, and this column has multi-row because all the three rows have the same state number. The second column Pid is the requester processor identification number in addition to the other idle processors IDs. The objective of displaying the other processors’ information is to show the consistency and readability of information within the table at every state. The third column is the operation indicator RW. The fourth and forth columns are the requested address Addr in decimal format and binary format respectively.
Figure 5.3: TEMPURA EXECUTION AT STATE 0
Figure 5.4: ANATEMPURA SIMULATION AT STATE 0
The length of the binary address is subjected to the space in the implementation within the runtime verifier AnaTempura. As the addresses’ values are generated randomly by using the modulo operation over 30, the highest value of the requested addresses is 29, which means that at least 5-bits length is adequate to represent the decimal addresses between 0 and 29 in binary format. The sixth column is the index of the cache. As explained earlier in Equation 5.2, the length of the index can determine the size of this index. For instance, in our case the length of the index bits is 3 which means $2^3 = 8$ Indexes. The seventh column is the Valid bit field which is an indicator whether the cache block in a specific index is valid or invalid. If the cache block has a datum in it, then the value of the valid bit is 1 which means true. Otherwise, it is 0 which means false. The eighth field is the Dirty bit which is an indicator of the consistency between a specific cache block and its correspondent in the main memory. If they are consistent, then the dirty bit value is 1 which means true. Otherwise it is 0 which means false. The ninth column is the Tag field which is the upper five portions of the requested address as the lower three portions are used for the index. Alternatively, the Tag value can be determined as a result of the division of the requested address over the length of the cache index as explained in Equation 5.3. The tenth column is the Hit-Miss which is the result of the requested address. When the requested address is found within the cache, it is either Read Hit or Write Hit, depending on the second column operation $RW$’s value. The eleventh column is Data where the value of the cache block for all the processors within the cache show their values. I set all the cache blocks for all the processors to an initial value $-8$ to avoid any execution error as I could not leave these cache blocks empty. They have to have integers number as values, as the language I used, Tempura, does not support null values. The twelfth column is the Coherence State; in other words it is the MSI Protocol criteria. For more about the MSI Protocol, refer to section 5.2.2. The thirteenth column is the value of the requested address within the main memory. Again, I set the integer number $-16$ for all the memory addresses as initial values for the same reason I used $-8$ as initial value for the cache blocks.
### CHAPTER 5. CASE STUDY: CACHE CONTROLLER

#### Table 5.10: Tempura Run of Interleaved Parallel Local Processors 0, 1 & 2

<table>
<thead>
<tr>
<th>State</th>
<th>Pid</th>
<th>Oper.</th>
<th>Addr$_{10}$</th>
<th>Addr$_{2}$</th>
<th>Cache[Index]</th>
<th>VBit</th>
<th>DBit</th>
<th>Tag</th>
<th>Hit-Miss</th>
<th>Data</th>
<th>MSI</th>
<th>Memory[Addr] = Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (0)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[110]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read Miss</td>
<td>−16</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>0 (1)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[110]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>0 (2)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[110]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>1 (2)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[010]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read Miss</td>
<td>−16</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>1 (0)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[010]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>1 (1)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10</td>
<td>Cache[010]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>1 (0)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>6</td>
<td>Cache[110]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Write Miss</td>
<td>13</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>1 (1)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>6</td>
<td>Cache[110]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>1 (2)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>6</td>
<td>Cache[110]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001110]=−16</td>
</tr>
<tr>
<td>2 (2)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Read Miss</td>
<td>−16</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>2 (0)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>2 (1)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001010]=−16</td>
</tr>
<tr>
<td>3 (1)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Cache[001]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Write Miss</td>
<td>14</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>3 (2)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Cache[001]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>3 (0)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Cache[001]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>4 (0)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>15</td>
<td>Cache[111]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Write Miss</td>
<td>19</td>
<td></td>
<td>Memory[0001111]=−16</td>
</tr>
<tr>
<td>5 (1)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>15</td>
<td>Cache[111]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001111]=−16</td>
</tr>
<tr>
<td>5 (2)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>15</td>
<td>Cache[111]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001111]=−16</td>
</tr>
<tr>
<td>6 (2)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>3</td>
<td>Cache[011]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Write Miss</td>
<td>14</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>6 (0)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>3</td>
<td>Cache[011]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>6 (1)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>3</td>
<td>Cache[011]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000011]=−16</td>
</tr>
<tr>
<td>7 (0)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>9</td>
<td>Cache[001]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Write Miss</td>
<td>3</td>
<td></td>
<td>Memory[0001001]=−16</td>
</tr>
<tr>
<td>7 (1)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>9</td>
<td>Cache[001]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Write Miss</td>
<td>14</td>
<td></td>
<td>Memory[0001001]=−16</td>
</tr>
<tr>
<td>7 (2)</td>
<td>1</td>
<td></td>
<td>1</td>
<td>9</td>
<td>Cache[001]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Write Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001001]=−16</td>
</tr>
<tr>
<td>8 (2)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Read Hit</td>
<td>−16</td>
<td></td>
<td>Memory[0000100]=−16</td>
</tr>
<tr>
<td>8 (0)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000100]=−16</td>
</tr>
<tr>
<td>8 (1)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>Cache[100]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0000100]=−16</td>
</tr>
<tr>
<td>9 (0)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>8</td>
<td>Cache[000]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read Miss</td>
<td>−16</td>
<td></td>
<td>Memory[0001000]=−16</td>
</tr>
<tr>
<td>9 (1)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>8</td>
<td>Cache[000]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001000]=−16</td>
</tr>
<tr>
<td>9 (2)</td>
<td>0</td>
<td></td>
<td>0</td>
<td>8</td>
<td>Cache[000]</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>Read Miss</td>
<td>−8</td>
<td></td>
<td>Memory[0001000]=−16</td>
</tr>
</tbody>
</table>
5.3.1.2 External Programs: Local Processors

Figure 5.5 shows the external programs’ outputs. These external programs are the processors which run randomly and independently. Each external program gets a Processor identification number \( Pid \) assigned to it within the cache controller program, for instance:

![Listing 5.1: Pids Assignment to Local Programs](image)

The lines in listing 5.1 are chunk of the global program written in Tempura language [182]. The global program loads the three local programs which respectively represent processor 0, 1, and 2. A function called \texttt{Prog\_send1()} is responsible for assigning the \( Pid \) to load a local program as external program. For instance, line 1 assigns \( Pid_0 \) to load a local program called \texttt{Processor\_0\_5}. This local program gets loaded by \( Pid_0 \) to accomplish its computation. In line 5, as \( Pid_0 \) was chosen to load \texttt{Processor\_0\_5}, the same \( Pid_0 \) requests to run a function within this local program, \texttt{Processor\_0\_5}, called \texttt{L2\_Processor\_0()}. Within this function, the main memory address requests are created and then inserted as assertions data to the global program to fulfill each processor request. Once the assertion data is received by the global program, the cache controller deals with the data according to the criteria described earlier in the previous sections to meet a set of properties of interest such as memory consistency and cache coherence. These properties are discussed in details later in this chapter.
Figure 5.5: Local States & Properties of Processors 0, 1, 2 at State 0
In the first line of each processor’s output window, it is noticeable that the status of the processor is either active or idle. When it is active, it shows the request information which is assigned to this processor. Otherwise, it shows that the processor is idle as illustrated in Figure 5.5.

5.3.1.3 Raw Data Analysis

The whole execution of a cache controller case study in Tempura/AnaTempura can be found in Appendix A. The raw data in Table 5.10 is identically copied from the execution in Appendix A. The number of columns in Table 5.10 is the same number in the execution plus a new column within the table. The new column is the state number column which is an indicator of the state number of the execution of cache controller. As Table 5.10 show, there are ten states, from 0 to 9. In each state, the data is displayed of the correspondent requested address within the cache of the three processors including the requester processor, or what is called the active processor, and the other idle processors. The purpose of displaying all information of processors is to increase the readability of the run and to show the validity of the MSI (or Coherence States) results. For instance, if the MSI column of processor X is Modified[X], then the data stored in the Cache and Memory columns of the requested address has to be inconsistent. The purpose of this check is to guarantee the memory consistency property. Another purpose is to guarantee the cache coherence property. The latter property concerns the cache coherence which is a discipline that maintains multiple cache blocks which share the same resource. For instance, if the cache block has data which is shared by another cache block of another processor, then the MSI (or Coherence States) changes the states of these cache blocks to Shared.

The first 3-multiple rows in Table 5.10 is state 0 of the cache controller. In the second column of the table, Pid, the first three rows represent the three processors identification numbers Pids. The Pid’s value has two digits, one digit outside the parentheses, and the other inside the parentheses. The digit outside the parentheses represents the requester Pid, while the digits
inside the parentheses represent the idle processors. The requester \textit{Pid} also includes itself as a \textit{Pid} inside the parentheses, and it always comes in the first row of each state run. For instance, in \textit{Pid}'s column, in the second row at state 0, the value 0(1) represents the requester's \textit{Pid} which is 0 and the idle \textit{Pid} which is 1 and so on. It is very important to mention that columns 6, 12 and 13 in Table 5.10 use square brackets. They are not numbered referencing styles.

**STATE 0:** Pid$_0$ creates a read request of address 14 in the memory as shown in table 5.12. The address 14$_{10}$, in binary format 0001110$_2$, gets assigned to the index 6$_{10}$ or alternatively 110$_2$ in binary format. Binary format is used in the case study. Therefore, Memory address 14 gets assigned to Cache[110]. The next column, VBit, is Valid Bit, and it is set to 0 as initial values. As the block cache[110] is empty, the previous value of it was 0. However, after the request is missed and the data of the correspondent address is fetched from the memory, this Valid Bit has now changed its value to 1. The initial values in the Dirty Bit column, DBit, are 0 as well, and as the data has just recently been fetched from the memory to the cache, this DBit stays 0. This means that the cache and the memory are consistent, while the other processors still hold their initial values although these values might mislead the reader and give a false impression about them. It is believed that the initial value of DBit is supposed to be -1 instead of 0 because 0 means that the cache block and the memory are consistent. However, as we can see in their correspondent Data column, these idle processors still have the initial values of the Data column, -8, and they obviously seem inconsistent with memory. The next column is the Tag column where the upper portion of the binary address is assigned to be tag value. The Tag value of address 0001110$_2$ is the four upper portions: 0001$_2$ or 1$_{10}$. The tag value can be determined using Equation 5.3. The tag value is used with the VBit value as a conditional conjunction to meet any hit requests, otherwise it is a miss regardless of the kind of operation it is. As the initial value of Tag is -1, the idle Pids holds -1. The Hit-Miss column is a result of a request created by an active processor of an address in the memory either to read from or write to. In this state, state 0, the request to read address 14 is missed because the private cache memory of Pid$_0$ has
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

not got that address stored in it which leads the memory to fetch the requested data of address 14 to the cache memory as seen in Table 5.12 and to the Pid₀ as seen in Table 5.12 to continue its computation. As a result of this behaviour, the next column, Data column, has -16 after it has been fetched from the memory. MSI column (or Coherence States) has set the state of Pid₀ to Shared[0] as it is consistent with the memory while the idle processors are still invalid. The Memory[Addr] column shows the data of the requested address created by the active processors.

STATE 1: Pid₂ is a requester processor in this state. It requests to read the memory address 10, binary 1010₂. This address is assigned to the cache block Cache[010]. The VBit was 0, and this request changed it to 1. The DBit stays 0 as the cache[010] and Memory[0001010] are consistent after the requested data of address 10 is fetched from the memory. The tag field of the Cache[010] of address 10 is 1. As the private cache block of Pid₂ has not got the requested address stored in it, the request then is Read Miss and the data of that address gets fetched from the memory as it is unavailable in other Pid’s cache blocks. Therefore, the coherence state of Cache[010] of Pid₂ is set to shared as it is consistent with the memory. The idle processors stay in invalid state as they still have initial values in their cache blocks, which means they are inconsistent with the memory.

STATE 2: Pid₀ requests to write data 13 to address 6. The cache block Cache[110] stores this data instead of the memory. The point of creating a cache memory reveals in this case where a write operation does not need to be done to the memory as writing to the cache is faster than writing to the memory. VBit and DBit have changed their values to 1. The most interesting change in this case is the DBit, where the new value 1 indicates that the cache block is dirty bit because the write operation occurs locally which means the memory has not been involved in this operation yet. Cache[110] and Memory[0000110] are inconsistent and this is why the Coherence State of Cache[110] is set to Modified[0]. Once this cache block Cache[110], the dirty case only, gets a future write to it by another address which shares the same cache index 110, the old data gets replaced and moved to its correspondent memory address, while
CHAPTER 5. CASE STUDY: CACHE CONTROLLER

the new data takes over index 110. For instance, address 6 shares index 110 with addresses
14, 22, 30 or any address that has 6 as a result of modulo operation of that address over the
length of the index which is in this case eight indexes. If address 22 writes a new value to
Cache[110] of Pid$_0$, then the old data which is 13 gets replaced and moved to Memory[0000110],
and the new data gets written to Cache[110] and sets DBit to 1 to indicate that Cache[110] and
Memory[0010110] are inconsistent. This mechanism is called Write-back. At state 0, address 14
was read and moved to Cache[110]; its value in the memory is -16 and has set DBit to 0 because
it was consistent with the memory. However, at this state, state 2, address 6 has written data
13 to Cache[110] after the replacement of the old data which is -16. The write-back operation
is not witnessed because DBit of Cache[110] was 0 before the replacement which means that
cache[110] and Memory[0001110] were consistent. Now we go back to the case study where the
value of Cache[110] is 13, while the value of its correspondent main memory Memory[0000110]
is -16, which is the initial value. The cache blocks of Pid$_1$ and Pid$_2$ stay invalid.

**STATE 3** Pid$_2$ requests to read address 4. The cache block, Cache[100], has not got the re-
quested address stored in it and, therefore, VBit is set to 1 after the request gets Read Missed.
The data of the correspondent address gets fetched from the memory to Cache[100] and, con-
sequently, DBit has been set to 0 due to the consistency between the cache and the memory of
the correspondent address. The Coherence State has changed the state of Pid$_2$’s cache block
Cache[100] to Shared[2]. Address 4 has 0 as the tag value.

**STATE 4:** Pid$_1$ requests to write data 14 to address 1. VBit and DBit change their values to
1 after the request gets Write Missed. Write-back operation is interesting mechanism, no write
operation occurs to the memory and, due to this, DBit’s value of cache block Cache[001] is
1. Consequently, the Coherence State changes the state of Cache[001] of Pid$_1$ to Modified[1].
Memory[0000001] is inconsistent with Cache[001] of Pid$_1$.

**STATE 5:** Pid$_0$ requests to write data 19 to address 15. Cache block Cache[111] stores the
new data and sets VBit and DBit to 1 to indicate that the cache block is valid and dirty at the
same time which eventually implies a write-back to the memory. As the requested address is 15, the tag value is 1. The request gets Write missed. The Coherence State is Modified[0]. The data of the correspondent cache block is 19, while in the memory still holds the initial value -16.

**STATE 6:** Pid$_2$ requests to write data 14 to address 3. Cache block Cache[011] stores the data of address 3 and sets VBit to 1. DBit changes its value to 1 to indicate that it is a dirty cache block and a write-back is eventually needed. The Tag value is 0. The request gets write missed as the cache block was invalid. The Coherence State is Modified[2] due to the inconsistency between the cache and the memory.

**STATE 7:** Pid$_0$ requests to write data 3 to address 9. Address 9 shares the same cache block Cache[001] with address 1. As the cache block Cache[001] has been written data 14 to it via Pid$_1$ at state 4, Pid$_0$ writes to its private cache block Cache[001] new data which is 3. Now there are two caches holding different data but sharing the same cache block. The most recent request by Pid$_0$, at this state, sets its DBit to 1 to indicate that it contains dirty data and at the same time, Pid$_0$ sets DBit of the cache block cache[001], which belongs to Pid$_1$, to 0 after the old data, data 14, gets written back to the memory. VBit is set to 1 as this cache block of the Pid$_0$ was invalid. As the address is 9, the tag value is 1. The request is a write miss because the VBit was invalid. The data in the cache block Cache[001] which belonging to Pid$_0$ is 3, while the one belongs to Pid$_1$ is 14. The Coherence State of Cache[001] that belongs to Pid$_0$ is Modified[0] because it is inconsistent with memory address 9. The Coherence State of Cache[001] that belongs to Pid$_1$ is set to Invalid[1].

**STATE 8:** Pid$_2$ requests to read address 4 again after it requested reading the same address at state 3. A read hit is encountered. The data is already fetched from the last read miss at state 3 to cache block Cache[100] which is -16. VBit and DBit stabilise their values, 1 for the former and 0 for the latter. The Coherence State stays Shared[2] as the cache block Cache[100] is consistent with the memory address Memory[0000100].
STATE 9: Pid0 requests to read memory address 8 Memory[0001000]. The private cache of Pid0 has not got this address stored in it, therefore, VBit sets its value to 1 and DBit sets its value to 0 after the data of address 8 is fetched from the memory which is -16. Tag value is 1. The Coherence State changes to Shared[0] because the cache block Cache[000] of Pid0 and Memory[0001000] are consistent; they both hold the same data. The same cache blocks of idle processors’ caches, Pid1 and Pid2, invalidate their cache blocks Cache[000] to Invalid[1] and Invalid[2].

To check all the above analysis, I refer the reader to Tables 5.12 and 5.12, and Figure 5.6. In addition to these tables and this figure, Appendix A has screen-shots of the implementation of the case study.

Table 5.11: REQUESTS OF Pid0, Pid1, & Pid2 RESPECTIVELY.

<table>
<thead>
<tr>
<th>State</th>
<th>Oper</th>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000000000000000000000001110</td>
<td>−16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00000000000000000000000000001110</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>00000000000000000000000000000110</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>00000000000000000000000000000110</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>00000000000000000000000000000111</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>00000000000000000000000000000100</td>
<td>−16</td>
</tr>
</tbody>
</table>

Table 5.12: L2 CACHE MEMORY OF Pid0, Pid1, & Pid2 RESPECTIVELY.

<table>
<thead>
<tr>
<th>Index V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>00000000000000000000000000000110</td>
<td>−16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>00000000000000000000000000001111</td>
<td>3</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>00000000000000000000000000000101</td>
<td>13</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>00000000000000000000000000000101</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>00000000000000000000000000000110</td>
<td>−16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>00000000000000000000000000001111</td>
<td>3</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>00000000000000000000000000000101</td>
<td>3</td>
</tr>
</tbody>
</table>
Figure 5.6: States & Intervals ($\sigma_{m}^{n}$, where $m$ is state number, $n$ is Processor id) of Cache Controller and Memory values
5.3.1.4 Properties Check of The Cache Controller

In this section, set of properties of interest are checked against the behaviour of the Cache Controller such as memory consistency and cache coherence. Memory consistency property guarantees that the data of a cache block and its correspondent copy within a memory address are consistent. While the cache coherence property guarantees that the cache blocks within multicore processor are subjected to MSI protocol to ensure the validity of these caches and their data. Table 5.13 illustrates the check of the correctness properties over all the states and for all cache memories and the main memory.

<table>
<thead>
<tr>
<th></th>
<th>Pid</th>
<th>Invalid State Prop.</th>
<th>Shared State Prop.</th>
<th>Consistency Prop.</th>
<th>MSI Protocol</th>
<th>Global State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Expected</td>
<td>Actual</td>
<td>Expected</td>
<td>Actual</td>
<td>Expected</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>NA</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pass</td>
<td>Pass</td>
<td>NA</td>
<td>Pass</td>
<td>Pass</td>
</tr>
</tbody>
</table>
5.4 Summary

In this chapter, a benchmark case study, Cache Controller, has been run. The results demonstrated that the proposed model can handle parallel/distributed systems significantly. The specification of a cache controller system is given in Interval Temporal Logic (ITL), while the runtime verification is given in Tempura language and implemented using AnaTempura. The set of properties of interest such as memory consistency and cache coherence are met, proved, and satisfied.
Chapter 6

Evaluation of Parallel Runtime Verification Framework (PRVF)

Objectives:

- To introduce MATLAB
- To link AnaTempura to MATLAB
- To produce Correctness Properties of the Cache Controller System
- To evaluate Parallel Runtime Verification Framework (PRVF)
- To present Discussion and Related Work
6.1 Introduction

In the previous chapter, Chapter 5, the case study of private L2 Cache Memory was designed, modelled, and implemented using the runtime verifier AnaTempura. The implementation has successfully met the expected behaviour of the system and fulfilled correctness properties set earlier. However, in this chapter I will run random and independent evaluation techniques using MATLAB as external tools in order to exclude any bias judgement upon the proposed model, Parallel Runtime Verification Framework (PRVF), with regards to its reliability, efficiency, performance, robustness etc.

In order to be able to use MATLAB for this purpose, a set of practical steps have to be applied to integrate AnaTempura, which is the primary tool for the implementation of the case study, with MATLAB. The integration step plays a primary role in order to completely allow AnaTempura to communicate natively with MATLAB. This communication between these two powerful tools will complement the process towards comprehensive evaluation techniques.

In this chapter, MATLAB and a brief description are given. Afterwords, I will explain in details how to integrate AnaTempura with MATLAB. An illustration of such integration using simple Tempura, Tcl, and shell scripts will serve as a basic understanding of the whole process of the evaluation techniques.

Once AnaTempura and MATLAB are integrated, I will import assertion data from AnaTempura during the runtime verification and pass these data to MATLAB in order to conduct evaluation techniques. After that, MATLAB produces a comprehensive evaluation based on the received data via the assertion data which were generated within AnaTempura. Then the evaluation outcomes and judgements of both tools, AnaTempura against MATLAB, are compared. If both evaluation outcomes and judgements are identical, the proposed model is reliable, efficient, performing and robust. If otherwise, vice versa, and a reconsideration of the proposed model is essential.
6.2 MATLAB

MATLAB is the acronym for MATrix LABoratory which was developed by MathWorks to serve as a multi-paradigm numerical computing environment and proprietary programming language. MATLAB supports the data to be represented as matrix in order to allow matrix manipulations, representing function and data in plots, algorithms implementation, interface creation, interacting and interfacing with other programming languages such as Java, C, C++, C#, Fortran and Python [164]. The advantage of supporting these common programming languages allows the users of other programming languages via shell scripts written in C, for instance, to integrate models built in programming languages not supported by MATLAB directly.

MATLAB has a package called Simulink which plays a primary role in graphical multi-domain simulation and model-based design for dynamic and embedded systems. In 2018, the number of users MATLAB exceeded three million worldwide from multiple disciplines [165].

Besides being a high-performance language, MATLAB has powerful features including modelling, analysing, and prototyping technical computation. MATLAB enables the computation to be natively expressed in mathematical notation which enhances the delivered solutions. Mainly, MATLAB is used for the following purposes: [7]

- Mathematics and Computation
- Algorithm Development
- Modelling, Simulation, and Prototyping
- Data Analysis, Exploration, and Visualization
- Scientific and Engineering Graphics
- Application Development, including Graphical User Interface building
6.3 Integrating MATLAB and AnaTempura

AnaTempura is a runtime verifier of systems using Interval Temporal Logic (ITL) and its executable subset Tempura. For more information about AnaTempura refer to Section 4.2, Chapter 4 as it is completely covered in this chapter. Tempura interpreter is programmed in C language. This makes it advantageous so an integration of MATLAB and AnaTempura can be done via Bourne shell scripts [39]. I prefix the shell by the author’s name Stephen Bourne, in order to distinguish it from other shell languages. However, in the next sections, I use only “shell” instead of “Bourne shell”. Shell scripts have the file extension “.sh” in which they are computer programs designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text[135].

6.3.1 Running MATLAB

MATLAB can be run using Microsoft Disk Operating System MS-DOS or Linux/macOS Terminals by typing the short command:

```matlab
matlab
```

This simple command runs MATLAB in the machine either as a desktop version or internally within the DOS or Terminal. The desktop version is the default option, alternatively, simply just add the flag “-desktop” to the previous command as follows:

```matlab
matlab -desktop
```

Otherwise, “-nodisplay -nodesktop” flags force MATLAB to run without the desktop GUI and run it internally whether within the MS-DOS, Terminal, or within other tools such as AnaTempura:

```matlab
matlab -nodisplay -nodesktop
```
Once MATLAB is run within external systems as Figure 6.1 illustrates, it offers all of its powerful features via loading MATLAB scripts, and executing MATLAB commands natively as they were being executed in MATLAB environment.

A matrix \( C \) of 1 row, 3 columns (\( 1 \times 3 \) matrix), which has these values \( C = [1, 2, 3] \) can be created and retrieved by typing the following script:

```
1 M=[1,2,3];
2 M % Display M
3 A=M(:,:); % Colon mark displays all rows, columns.
4 A
5 B=M(1,2); % Display the cell at row 1, column 2.
6 B
```

![Figure 6.1: Running MATLAB](image)

![Figure 6.2: MATLAB Script](image)
The percentage mark “%” is used to comment. Figure 6.2 illustrates the outputs after the execution of this short script using macOS Terminal.

MATLAB can be run as well via a shell script. For instance, the shell script in Listing 6.2 runs MATLAB first then runs a MATLAB script as Listing 6.3 illustrates:

```bash
#!/bin/sh
matlab -nosplash -nodesktop -r "run('Arithmetic.m');"
```

The above shell script first runs MATLAB in non-desktop GUI mode, while MATLAB script “Arithmetic.m” presents how arithmetic operations can be done in MATLAB language as follows:

```matlab
X=10; Y = 2;
fprintf('%d * %d = %d
',X,Y,X*Y);
fprintf('%d / %d = %d
',X,Y,X/Y);
fprintf('%d + %d = %d
',X,Y,X+Y);
fprintf('%d - %d = %d
',X,Y,X-Y);
fprintf('%d mod %d = %d
',X,Y,mod(X,Y));
```

Figure 6.3 illustrates the output for the execution of MATLAB script in Listing 6.3 after being called via the shell script in Listing 6.2:

```
MATLAB (R) 9.6.0.1072631 (macOS) 64-bit
Copyright 1984-2018 The MathWorks, Inc.
To get started, type doc.
For product information, visit www.mathworks.com.
10 * 2 = 20
10 / 2 = 5
10 + 2 = 12
10 - 2 = 8
10 mod 2 = 0
```

Figure 6.3: MATLAB Arithmetic Script Output
This short tutorial of how MATLAB is run and how to run scripts written in MATLAB is for demonstration sake. For more information and tutorials visit MATLAB website [164].

6.3.2 AnaTempura Runs MATLAB

AnaTempura can run external programs or systems by annotating the name of these programs or systems written in different languages. This annotation then executes whatever is written inside these programs or shell scripts. Every annotation is assigned to a unique process identification \( Pid \), so they do not clash or delay the execution time. For instance, the annotation within a Tempura program in Listing 6.4, run by AnaTempura, calls Tcl, Java, C programs and shell script as external programs:

Listing 6.4: Annotation within Tempura Program

```
1 Tempura code . . .
2 /* tcl Cache 0 */
3 /* java Hello 1 */
4 /* prog Fac 2 */
5 /* prog Script.sh 3 */
6 Tempura code . . .
```

These markers “/*” and “*/” are used respectively to open and close comments. However, the texts between these markers are sometimes executable in case they are prefixed by keywords such as tcl, java, prog; in these cases, they call external programs independently. Processes \( Pid_0 \), \( Pid_1 \), \( Pid_2 \) and \( Pid_3 \) are assigned to Tcl, Java, C programs and Shell script respectively. Process \( Pid_3 \) calls the shell script to be executed. The shell script runs MATLAB as explained above, and it also runs the MATLAB script which is already created to do some specific computations. The execution of these various programming languages and shell scripts enriches AnaTempura and empowers it to be used widely.

The integration procedures are going to be thoroughly explained and demonstrated in this section. First of all, a Tempura program has to be created as Listing 6.5 illustrates:
Chapter 6. Evaluation of Parallel Runtime Verification Framework (PRVF)

Listing 6.5: Tempura Program “Hello.t”

1 load "../../../library/conversion".
2 load "../../../library/exprog".
3 load "../../../library/tcl".
4 /* tcl Hello 0 */
5 /* prog Hello.sh 1 */
6 set print_states = false.
7 define Send_To_MATLAB(C) = {
8    tcl("init", [C]) and
9        always tclbreak()
10    }.
11 /* run */ define Test() = exists C: {
12    input C and output C and len(0) and Send_To_MATLAB(C)
13    }.

Annotations are made in line 4 and 5. Annotation in line 4 calls a Tcl program “Hello.tcl” as Listing 6.6 illustrates, and the process assigned to executed it is Pid_0, while the annotation in line 5 calls the shell script “Hello.sh”, as Listing 6.7 illustrates, and the execution of the shell script is assigned to process Pid_1. The Tempura program in Listing 6.5 initialises a state variable called “C”. The value of this state variable is entered via AnaTempura monitor at runtime, and it has to be suffixed by a dot mark “.” in order to carry on the execution and receive the input of “C”. Without the dot mark, AnaTempura monitor waits until doing so.

Line 12 in Listing 6.5 prompts the input to be entered and assigned to “C”, then outputs the entered values. Afterwards, it passes the entered values to function Send_To_MATLAB(C). This function is declared in line 7 of the same Listing. The function carries the input values of “C” and a connection with an external program written in Tcl is initialised as line 8 illustrates. The command tcl("init", [C]) passes the input values received via AnaTempura monitor to a procedure init within a Tcl external program, in Listing 6.6, which has been already called via annotation command in line 4 of the same Listing. The inputs that were entered as Figure 6.4 illustrates are “36.”, “37.” and “38.”.

136
CHAPTER 6. EVALUATION OF PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

Figure 6.4: AnaTempura inputs numbers to file “input.txt”

The following Listing 6.6 is a Tcl program “Hello.tcl”; it receives the input values entered within AnaTempura monitor. The procedure `proc init {nl}` in line 2 is the mediator as it was called within the previous Listing 6.5. The command in line 3 sets a list of index 0 in order to assign the received inputs to “C”. Lines 4 to 6 create a text file “input.txt” then write to and read from this file.

Listing 6.6: Tcl Program “Hello.tcl”

```tcl
namespace eval ::out {

proc init {nl} {
    set C [lindex $nl 0]

    set fp [open "input.txt" a+]
    puts $fp "$C"
    close $fp
}
```
Once the writing process is done, the text file stores the input values entered to it via the AnaTempura monitor as Figure 6.5 illustrates:

![Figure 6.5: File Content for “input.txt”](image)

At this point, the integration step is reached. The annotation in Listing 6.5, line 5 is entitled to execute the shell script in Listing 6.7. The script runs MATLAB in a non-desktop GUI mode, which means MATLAB will be loaded into AnaTempura monitor windows as external program of *Pid₁* because this process has been assigned within the annotation in line 5. The flag “-r” in Listing 6.7, line 2 indicates that the following text enclosed in double quotations is a MATLAB code and has to be executed. Alternatively, a MATLAB script with the same code could be loaded instead of writing a MATLAB code within the shell script, but this short MATLAB code is meant to load the input text file “input.txt” which is already created by AnaTempura external program; it then retrieves that inputs entered in the text file. Listing 6.7 illustrates shell script “Hello.sh”:

Listing 6.7: Shell Runs MATLAB & Load Data from “input.txt”

```bash
#!/bin/sh
matlab -nosplash -nodesktop -r "load('input.txt'); C=input(:,1)"
```

Figure 6.6 illustrates that MATLAB has been run externally within AnaTempura, and the inputs values entered into AnaTempura are delivered successfully to MATLAB, and they can be manipulated, analysed, simulated, etc.
The work of integrating AnaTempura with MATLAB is crucially important and original. Such an integration bridges the gap between runtime verification tools and MATLAB which offers a variety of toolboxes such as Model-Based Design Simulink, Fuzzy Logic, Robotics System, Aircraft Intuitive Design (AID), Statistics and Machine Learning and much more. MATLAB is trusted by millions of engineers, scientists, companies, industrials, institutions, universities, etc.. This diversity of applications is promising in the way that AnaTempura and MATLAB can play a great role together. The benefits are mutual for either systems, and they push each other’s limitations.

By now, AnaTempura and MATLAB are integrated and can communicate natively. The next section sheds light on correctness properties of interest such as Memory Consistency and Cache Coherence State of MSI protocol. At runtime, these correctness properties have to be verified that they are met, proved, and satisfied. In order to perform this task the integration betweenAnaTempura and MATLAB have to be done to be able to compare the outcomes of AnaTempura and MATLAB.
6.4 Correctness Properties

According to Berkovich et al. [29], in computing systems, Correctness refers to the assertion that a system satisfies its specification. The system I used for the case study in Chapter 5 is a Private L2 Cache Memory of multi-core processor architecture. The proposed model, Parallel Runtime Verification Framework (PRVF), has implemented the case study successfully. However, random and independent evaluation techniques are intended to be applied using MATLAB in order to exclude any bias judgement upon the proposed model using only AnaTempura. MATLAB is going to be used to produce another version of the judgement, and if both judgements are identical, the proposed model is then reliable, efficient, performing and robust.

Memory Consistency and Cache Coherence State of MSI Protocol properties are the correctness properties I intended to investigate. Each correctness property will be defined at first and then expressed formally in formal-based framework, Interval Temporal Logic (ITL). After that, AnaTempura will run the case study using the proposed model, PRVF, in order to monitor the behaviour of the system under scrutiny. Once AnaTempura runs the check of the system, MATLAB will be run by AnaTempura, and they will communicate with each other and exchange assertion data. AnaTempura makes its judgement, and simultaneously MATLAB makes its judgement too. The judgements checks are then compared and analysed using MATLAB toolboxes.

6.4.1 Revisiting The Case Study of Cache Controller

The case study was described and implemented thoroughly in the previous chapter, Chapter 5. In this section I will only run the case study using AnaTempura following the same steps applied earlier in the previous chapter; however, this time MATLAB is fully integrated with AnaTempura and will produce plotted charts representing the assertion data exchanged between AnaTempura and MATLAB. Screen shots of the case study being run are illustrated in Figures 6.7 & 6.8. Figure 6.7 displays the outcomes of the implementation of the case study using AnaTempura. The outcomes are formatted in a table where they are explained and analysed in Chapter 5.
CHAPTER 6. EVALUATION OF PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

Figure 6.7: AnaTempura Run of L2 Cache Memory of Processors 0, 1 & 2

Figures 6.8 demonstrates the outcomes generated in the table of Figure 6.7 in order to visualise the outcomes and increase the understanding of the case study in addition to monitoring purposes.
AnaTempura can now run MATLAB natively as an external program as Figure 6.9 illustrates. These external windows of external programs of processors 4 & 5 are responsible for running two shell scripts annotated within Tempura program which is written to run the case study. The figure shows that MATLAB has been run successfully by AnaTempura in order to monitor and analyse the behaviour of the cache memory.
Figure 6.9: External Programs of AnaTempura for Processors 4 & 5

Figure 6.10 represents the assertion data file produced at runtime by the cache implementation and at the same time MATLAB loads this file and reads the data inside the file in order to do the analysis step.

Figure 6.10: Dual Core Dual Processor System

All the source codes of the implementation of this case study are written in Tempura, Tcl, shell and MATLAB languages can be found in Appendix E. With regards to the produced out-
comes of the MATLAB scripts executed during runtime of AnaTempura, they are displayed in the next sections, Section 6.4.2 & 6.4.3.

6.4.2 Memory Consistency Property

A memory of a particular address is consistent if it holds the same value of at least one cache memory of the correspondent index. Otherwise, a memory is inconsistent. In the later case, a verdict of either true or false of a marker called dirty bit within cache memory architecture is switched accordingly. The dirty bit of the cache memory is true when cache memory and main memory of a correspondent address are inconsistent, otherwise, it is false. The correctness property of Memory Consistency can be formalised in Interval Temporal Logic (ITL) as follows:

$$\vdash \text{Memory}[\text{Addr}] = \text{Cache}[X][\text{Index}] \lor \text{Cache}[Y][\text{Index}] \lor \text{Cache}[Z][\text{Index}]$$

A memory of address $\text{Addr}$ has to be equivalent with at least one of cache memory indexes, $\text{Index}$ of processor $X$, $Y$ or $Z$. If the above formula is met, then a memory is consistent.

Now I will show the outcomes of the execution of MATLAB scripts that have been run within AnaTempura. A shell script has been annotated within a Tempura program; this shell script “CheckProperty1.sh” is responsible for running the MATLAB and then executing MATLAB script “Property1.m”. The MATLAB script gathers and assigns the assertion data being created during the runtime of AnaTempura, and then plots these data in graphs for each state of the execution of the case study.

The number of states is 10, from state 0 until 9. Each single state is individually captured by this MATLAB script and representing data of every single state of the related rows and columns of the assertion data is in the text file as Figure 6.10 illustrates. For referencing the addresses of
main memory and the indexes of the cache memory, I use this format:

\[
\begin{align*}
\text{Memory}[\text{Addr}] & = \text{Data} \\
\text{Cache}[\text{Pid}][\text{Index}] & = \text{Data}
\end{align*}
\]

Where,

**Addr**: indicates the requested address

**Pid**: indicates the processor identification

**Index**: indicates the entry within the cache of **Pid**

**Value**: indicates the data integer values

Noticed that, all the cache indexes is -8, while the main memory addresses data values is -16. Holding data -8 for the cache indexes means that cache indexes are empty and have no data yet. In contrast, a main memory holding data -16, means that the main memory is occupied and has data.

**STATE 0**: By referring to Figures 6.7 & 6.8, it can be found that the active processor is \textit{Pid\textsubscript{1}} and always comes as the first graph of the plot, at top-left corner, while \textit{Pid\textsubscript{2}}, \textit{Pid\textsubscript{0}} are considered idles. In this state, the \textit{Pid\textsubscript{1}} is the processor which requests to access address 14, \textit{Addr} : 14. The requested address fetches its data to the correspondent cache index of \textit{Pid\textsubscript{1}} in case the access is for reading \(RW = 0\) (refer to Figure 6.7).

When the access is for writing, \(RW = 1\), then the processor writes directly to the correspondent cache index, \textit{Index}[6], without updating the main memory. As the data of the main memory of the requested address is not updated yet and is different from the data of the correspondent cache index, the memory at state 0 is inconsistent. Once another processor requests the same address, this particular cache of \textit{Pid\textsubscript{1}} fetches data 25 to that processor. When this particular cache index of \textit{Pid\textsubscript{1}} is replaced by another request, the old data gets copied into the correspondent main memory and becomes consistent. See Figure 6.11.
Figure 6.11: Memory Consistency Check at State 0
STATE 1: Processor $\text{Pid}_1$ writes to index 6 the data 0, $\text{Cache}[1][6] = 0$. The correspondent address of this cache index, $\text{Memory}[6]$, is inconsistent as it has the data -16. Interestingly, at the previous state the old data of $\text{Cache}[1][6]$ was 25 and because it is replaced by a new data 0 in this state, the old data gets moved to its correspondent memory address $\text{Memory}[14]$. The main memory of address 6 $\text{Memory}[6]$ is inconsistent. See Figure 6.12.

![Cache of Pid 1 at State 1](image1)

![Cache of Pid 2 at State 1](image2)

![Cache of Pid 0 at State 1](image3)

![Main Memory at State 1](image4)

Figure 6.12: Memory Consistency Check at State 1
STATE 2: Process $Pid_2$ requests to read address 2 and because address 2 is not available in the cache index of all the three processors, the main memory fetches the data of address 2 to the requester processor. The main memory of address 2, $Memory[2]$, and the cache index of processor $Pid_2$, index 2, $Cache[2][2]$, have the same data, therefore, the main memory is consistent. See Figure 6.13.

Figure 6.13: Memory Consistency Check at State 2
STATE 3: Processor Pid₂ writes a new data 0 to index 0 which is the correspondent entry of the requested address 0. The cache index of Pid₂ is index 0, Cache[2][0] = 0, while the main memory of address 0 has different data Memory[0] = −16. Therefore, the main memory is inconsistent because it has not been updated yet. See Figure 6.14.

Figure 6.14: Memory Consistency Check at State 3
STATE 4: Address 15 is requested to be read by processor $P_{id_0}$. Because the cache index of the correspondent address is empty, the main memory of the requested address 15 fetches its data, $Memory[15] = -16$, to cache index 7 of $P_{id_0}$ as follows $Cache[0][7] = -16$. The main memory of address 15 is consistent because it has at least one cache holding the same data. See Figure 6.15.

Figure 6.15: Memory Consistency Check at State 4
STATE 5: Processor Pid$_2$ requests to read address 0 which is recently modified and written to at state 3 by the same processor, Pid$_2$. The data written to this cache index 0 is 0, therefore, cache index 0 of Pid$_2$ is Cache$_{2}[0] = 0$. The main memory of the requested address 0 is still not updated Memory$_{0} = -16$, therefore, the main memory is inconsistent. See Figure 6.16.

Figure 6.16: Memory Consistency Check at State 5
STATE 6: The read request is initialised by processor $Pid_1$ to read address 4. As cache index 4 is empty, the main memory fetches data -16 to this cache index. Cache index 4 of processor $Pid_1$ is $Cache[1][4] = -16$, and the main memory of address 4 is $Memory[4] = -16$. Consequently, the main memory is consistent. See Figure 6.17.

Figure 6.17: Memory Consistency Check at State 6
STATE 7: Processor Pid\textsubscript{2} requests to write to address 4 a new data 23. Index 4 of Pid\textsubscript{1} from the previous state has got data -16 fetched by the main memory. At this state, index 4 writes new data by Pid\textsubscript{2}. Now there are two different data $\text{Cache}[1][4] = -16$ and $\text{Cache}[2][4] = 23$. Processor Pid\textsubscript{2} is the most updated, while Pid\textsubscript{1} is outdated at this state. Therefore, the main memory of address 4 $\text{Memory}[4] = -16$, is outdated, and it is inconsistent. See Figure 6.18.

Figure 6.18: Memory Consistency Check at State 7
STATE 8: Processor $P_{id_2}$ requests to read address 5. Because cache index 5 of processor 2 $\text{Cache}[2][5]$ is empty, the main memory of address 5, $Memory[5] = -16$, fetches its data to the cache index, so it becomes $\text{Cache}[2][5] = -16$. As the cache memory of processor $P_{id_2}$ and the main memory have the same data, the main memory is consistent. See Figure 6.19.

Figure 6.19: Memory Consistency Check at State 8
CHAPTER 6. EVALUATION OF PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

STATE 9: Processor \( Pid_1 \) requests to read address 9. Cache index 1, \( Cache[1][1] \), of this correspondent address is empty, therefore, the main memory of address 9 fetches it data \( Memory[9] = -16 \) to this cache index, so it becomes \( Cache[1][1] = -16 \) which means that the main memory is consistent. See Figure 6.20.

Figure 6.20: Memory Consistency Check at State 9
6.4.3 Cache Coherence Property

The cache memory is coherent if it maintains one of the cache coherence protocols such as MSI, MESI, MOESI and many others [213]. The MSI Protocol is chosen because it is simple, and it serves the purpose. In section 5.2.2 I have covered the protocol and explained the meaning of the states the protocol indicates such as Modified, Shared and Invalid. Each state is shortened to one capital letter “M” for Modified, “S” for Shared, and “I” for Invalid. The following table, Table ??, describes the allowed and forbidden occurrences of these MSI states of the cache memory in multi-core architecture:

<table>
<thead>
<tr>
<th>Modified</th>
<th>Shared</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Invalid</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

This criteria is applicable on at least two entities or more. The Modified state “M” is highly restricted, and it does not accept any other states but Invalid “I”. The Shared state “S” is less restricted; it accepts another cache block to be either Shared “S” or Invalid “I”. The Invalid state “I” is tolerating, and it accepts all three states Modified “M”, Shared “S” or Invalid “I”.

Cache coherence can be achieved by maintaining MSI Protocol. This correctness property can be expressed formally in Interval Temporal Logic (ITL) as follows:

\[
\models \text{MSI Protocol}[X,Y] = (\text{State}[X][\text{Index}] = \text{Modified} \land \text{State}[Y][\text{Index}] = \text{Invalid}) \lor \\
(\text{State}[X][\text{Index}] = \text{Shared} \land (\text{State}[Y][\text{Index}] = \text{Shared} \lor \text{State}[Y][\text{Index}] = \text{Invalid})) \lor \\
(\text{State}[X][\text{Index}] = \text{Invalid} \land (\text{State}[Y][\text{Index}] = \text{Modified} \lor \text{State}[Y][\text{Index}] = \text{Shared} \lor \\
\text{State}[Y][\text{Index}] = \text{Invalid}))
\]
The formula expresses the allowed the MSI Protocol of two cache blocks for processors $X$ and $Y$. When the MSI Protocol states meet this formula, then the correctness property of the cache coherence is satisfied.

The acronym of MSI Protocol states “M”, “S” and “I” are replaced by “1”, “2” and “3” respectively in MATLAB graphs in order to be able to plot them as integer values of y-axis, while x-axis represents the processors identification $P_{id0,1,2}$.

**STATE 0:** By referring to Figures 6.7 & 6.8, it can be seen that processor $P_{id1}$ modifies cache block 6 by writing data 25 to it, so it becomes Modified $Cache[1][6] = 25$ because the main memory is not updated yet and no other cache blocks share this new data. The same cache blocks of processors $P_{id0}$ & $P_{id2}$ are still empty, and are, therefore, Invalid. See Figure 6.21

![Figure 6.21: Cache Coherence & MSI Protocol Check at State 0](image)
STATE 1: Processor \( Pid_1 \) writes a new data to the same cache index 6 which is written to in the previous state. The data is 0, \( Cache[1][6] = 0 \), MSI state of this cache index is Modified as neither the main memory nor the other cache blocks hold the new written data. The other processors \( Pid_0 \) & \( Pid_2 \) are still Invalid. See Figure 6.22

Figure 6.22: Cache Coherence & MSI Protocol Check at State 1

158
STATE 2: Processor $P_{id}2$ requests to read the main memory address 2. Cache index 2 of processor 2 $Cache[2][2] = -16$ and the main memory of address 2 $Memory[2] = -16$ share the same data. Therefore, the MSI protocol of $Cache[2][2]$ is Shared, while the other processors are still empty, which means that their MSI states are Invalid. See Figure 6.23
STATE 3: Cache index 0 of processor 2 \( \text{Cache}[2][0] \) is Modified because a write request is made. The data in main memory of the correspondent address is different from this cache index. The other processors \( Pid_0 \) & \( Pid_1 \) are empty, therefore, their MSI state are Invalid. See Figure 6.24

Figure 6.24: Cache Coherence & MSI Protocol Check at State 3
STATE 4: Processor $P_{id_0}$ requests to read a correspondent cache index 7 of the requested address 15, $Cache[0][7]$. The main memory $Memory = -16$ fetches its data to this cache index. Therefore, the MSI state of this cache block is Shared as it is consistent with the main memory data. The other processors are empty, and their MSI states are Invalid. See Figure 6.25

Figure 6.25: Cache Coherence & MSI Protocol Check at State 4
STATE 5: Processor $P_{id_2}$ requests to read address 0, because this address has recently been in state 3 and received a write request of data 0 to it. Therefore, at this state the requested read address returns 0, $Cache[2][0] = 0$. The MSI state is still Modified because the correspondent address in the main memory holds different data. The other processors, $P_{id_0}$ & $P_{id_1}$, are Invalid.

Figure 6.26: Cache Coherence & MSI Protocol Check at State 5
STATE 6: Cache block 4 of processor 1 holds data -16 after it is fetched by the main memory of address 4 as a consequent of the read request initialised by $\text{Pid}_1$. Therefore, $\text{Cache}[1][4] = -16$ which means that the MSI state of this cache block is Shared. Because the other processors $\text{Pid}_2$ & $\text{Pid}_0$ are still empty, their MSI states are Invalid. See Figure 6.27.

Figure 6.27: Cache Coherence & MSI Protocol Check at State 6
STATE 7: Processor $Pid_2$ requests to write data 23 to cache index 4, so it becomes $Cache[2][4] = 23$. This cache block of $Pid_2$ was Invalid in the previous state because it was empty. At this state, it becomes Modified as it has just received a new data while the main memory of the correspondent address is still not updated. Processor $Pid_1$ changes its cache block from being Shared at the previous state to Invalid at this state. Processor $Pid_0$ is still empty, therefore, it is Invalid too.

Figure 6.28: Cache Coherence & MSI Protocol Check at State 7
STATE 8: Processor $Pid_2$ requests to read cache index 5 and because this cache block is empty, the main memory of address 5 fetches its data, $Memory[5] = -16$ to it. The cache memory of processor 2 becomes $Cache[2][5] = -16$, which means it is in the Shared MSI state. Processors $Pid_0$ & $Pid_1$ are still empty, which means that their MSI states are Invalid.

Figure 6.29: Cache Coherence & MSI Protocol Check at State 8
CHAPTER 6. EVALUATION OF PARALLEL RUNTIME VERIFICATION FRAMEWORK (PRVF)

STATE 9: Processor $\text{Pid}_1$ requests to read cache index 1, and because this cache block is empty, the main memory of address 9 fetches its data, $\text{Memory}[9] = -16$ to this cache block, so it becomes $\text{Cache}[1][1] = -16$. Therefore, this cache block has Shared MSI state while the other processors, $\text{Pid}_0$ & $\text{Pid}_2$, have Invalid MSI states. See Figure 6.30

Figure 6.30: Cache Coherence & MSI Protocol Check at State 9
6.5 Discussion

The implementation of the case study of Cache Controller is deployed in order to measure the suitability and generality of the proposed model, Parallel Runtime Verification Framework (PRVF). As the proposed model is built and modelled using Tempura language, the subset execution version of Interval Temporal Logic (ITL) specification notation language, AnaTempura is used to run the case study in order to judge the model. I have illustrated the implementation of the case study and consequently the judgement of the used model in the previous chapter, Chapter 5.

In this chapter, I have deployed MATLAB to measure and judge the proposed model using AnaTempura. The integration steps have been covered thoroughly and illustrated visually in multiple figures. MATLAB gives the same judgement as the AnaTempura tool has given of the implementation of the case study using the proposed model PRVF.

The data analysis of ten states of two correctness properties is given in addition to twenty figures, Figures 6.11 to 6.30. These data analysis and illustrations figures produced by MATLAB prove that the proposed model is reliable, efficient, performing and robust.

The proposed model, PRVF, offers four mechanisms of implementation based on communication, concurrency and execution preferences. These four different mechanisms were introduced in Chapter 3. Although two of these mechanisms are implemented in this PhD thesis, which are Shared-Varibale Interleaving Concurrency and Shared-Varibale True Concurrency, the implementation of the other mechanisms will lead to success and the same judgement of this mechanism.

The other mechanisms are Shared-Variable True Concurrency, Message-Passing Synchronous Execution and Message-Passing Asynchronous Execution. These mechanisms are formally expressed in algorithms and are also modelled in flowchart figures. For algorithms refer to Algorithms 3, 4, 5 respectively and for the flowcharts refer to Figures 3.8, 3.9, 3.10 respectively.
For instance, Shared-Variable True Concurrency mechanism varies from the one implemented in this PhD thesis. It runs multiple systems simultaneously, and it has to maintain the shared variable value and apply an explicit synchronisation and control mechanisms such as monitors [120], semaphores [79], atomic operations and mutual exclusion (mutex) in order to provide a consistent shared variable value. Message-Passing (A)synchronous Execution are uses Message-Passing for communication. The only difference between these two latter mechanisms is the execution preferences. Asynchronous Execution allows parallel systems to have different timing clocks; for instance, one system could start and finish its execution differently compared to another parallel system running at the same time. Synchronous Execution restricts the timing clocks for systems run in parallel and force these systems to start and finish their executions at the same moment. In other words, Asynchronous Execution has different timing clocks, while Synchronous Execution has identical timing clocks.

### 6.6 Related Work

Although the proposed model, Parallel Runtime Verification Framework (PRVF), is dedicated to handle parallel computing systems at runtime, it is suitable to be a generic model for parallel computing regardless of being deployed at runtime. This is due to the fact that it considers the fundamental aspects of parallelism at software and hardware levels.

There are several models for parallel computing such as Parallel Random Access Machine (PRAM), Parallel Memory Hierarchy (PMH), Bulk Synchronous Parallel (BSP) and LogP. Each of these models has pros and cons, and then an explanation of the reason I favour in the proposed model, PRVF, over the other parallel computing models follows.

In 1978, Fortune and Wyllie [94] proposed the Parallel Random Access Machine (PRAM) as a natural evolution of the classic Random Access Machine (RAM) model. Ever since, PRAM model is considered to be one of the most used models for parallel computing in general and for parallel algorithms and analysis specifically.
In the 1990s, the PRAM model was considered an unrealistic model for parallel algorithm design and analysis due to the fact that at time that simultaneous operations could not offer constant memory access times by computers. The implementation of PRAM model was not complex as its design algorithms were suggesting. However, the General Purpose Graphics Processing Unit (GPGPU) computing Application Programming Interfaces (APIs) was introduced in 2006 and consequently the model, PRAM, became relevant.

PRAM model has different four variations in order to make modelling parallel algorithms more realistic. These variations are a $2 \times 2$ matrix of two sets: \{Exclusive,Concurrent\} and \{Read,Write\}. These variations, therefore, are Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), Exclusive Read Concurrent Write (ERCW) and Concurrent Read Concurrent Write (CRCW). These four variants are thoroughly explained in [205]. Concurrent writes have to meet one of the following protocols: 

1. Common, where all processors write the same value,
2. Arbitrary, where only one write is successful, the others are not applied,
3. Priority, where priority values are given to each processor (e.g., rank value), and the processor with the highest priority will be the one to write,
4. Reduction, where all writes are reduced by an operator (add, multiply, OR, AND, XOR). PRAM uses the shared memory model.

Alpern et al. [9] proposed Parallel Memory Hierarchy (PMH) model in 1993. The model was proposed to overcome the drawback of the PRAM model with regards to the constant time memory operations. Central Processing Units (CPUs) have memory hierarchies of registers, L1, L2 and L3 caches such as Intel Xeon E5 series and AMDs Opteron 6000 series. GPUs as well have registers, L1, L2 caches and global memory as a memory hierarchy such as Nvidia GTX 680 or AMDs Radeon HD 7850. Such memory hierarchies should be considered in the process of designing parallel algorithms.

A hierarchical tree of memory modules is used to define the PMH model. The processors are represented using the leaves while memory modules are represented using internal nodes. The more memory modules get closer to the processors, the faster, yet smaller, they become. On the
other hand, the more memory modules get far from the processors, the slower, yet larger, they become.

Uniform Parallel Memory Hierarchy (UPMH) is a simplified version of PMH model, and it is easier to model an algorithm than use PMH itself. UPMH model complements other models such as PRAM and Bulk Synchronous Parallel (BSP). (U)PMH uses the shared memory model.

Leslie Valiant [277] introduced in 1990 a parallel computing model, the Bulk Synchronisation Parallel (BSP), with primary consideration of communication aspects. The model highly considers synchronisation and communication where a number of processors with fast local memory are connected via a network. The processors can communicate easily and send or receive messages between each other. The algorithm which is used to build BSP model is called super-step, where it consists of three steps as a parallel block of computation: i) local computation, where \( p \) is processors perform up to \( L \) local computations, ii) global communication, where processors can send and receive data among them, iii) barrier synchronization waits for all other processors to reach the barrier. BSP uses the message passing model.

Culler et al [67] proposed the LogP model in 1993. LogP and BSP both consider the communication aspects by focusing on modelling the cost of communication a set of distributed processors. The cost of local operations in LogP is one unit of time, while the network considers latency \( (L) \), overhead \( (o) \), gap \( (g) \), and processors \( (P) \). The latency for communicating a message contains a word from the source to the target processor. Overhead can be measured by the amount of time a processor needs to send or receive. Gap can be measured by the minimum amount of time between successive messages in a given processor, while processors are the number of processors.

Latency, overhead and gap are measured in cycles. LogP is synchronised by pairs of processors, while BSP uses global barriers of synchronisation. LogP considers a message overhead, while BSP does not. Therefore, the determination of which model to use depends on the need for local or global synchronisation and whether the communication is overhead. LogP uses the
Comparing these parallel computing models to the proposed model in this PhD thesis, Parallel Runtime Verification Framework (PRVF), shows that PRVF considers both communication aspects such as the shared memory and message passing models unlike PRAM and PMH where they only use shared memory model. On the other hand, BSP and LogP use the message passing model in their algorithm designing and analysis. All models take into account two forms of concurrency, true and interleaving concurrency. In regards to (a)synchronous execution manners, PRAM and (U)PMH use asynchronous execution manner, while BSP and LogP both use (a)synchronous execution manner.

Table 6.2: Parallel Computational Models

<table>
<thead>
<tr>
<th></th>
<th>PRAM</th>
<th>(U)PMH</th>
<th>BSP</th>
<th>LogP</th>
<th>PRVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Concurrency</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interleaving Concurrency</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Message Passing</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Synchronous Execution</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asynchronous Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.2 illustrates a comparison between parallel computing models and the proposed model which is Parallel Runtime Verification Framework (PRVF). The comparison shows that PRVF fulfils all the aspects in the above table, while PRAM and (U)PMH models do not fulfil message passing communication programming model and synchronous execution manner. On the other hand, BSP and LogP models fulfil all aspects except the communication programming model of the shared memory. The comparison in this table shows the comprehensiveness of Parallel Runtime Verification Framework (PRVF) compared to the other parallel computing models.
6.7 Summary

MATLAB has been introduced in order to be used for evaluation purposes. MATLAB and AnaTempura have been linked and successfully communicated via exchanging assertion data. A demonstration has been given of how AnaTempura can run MATLAB natively. Correctness properties have been modelled formally and described in order to fulfil them. The behaviour of the cache controller case study has been detected and checked at runtime. MATLAB has captured the behaviour of cache controller by collecting the assertion data being generated by the model. The judgement upon the proposed model is made and a discussion and related work have been done as well. A comparison between parallel computing models and the proposed model is presented.
Chapter 7

Conclusion

Objectives:

• To present Thesis Summary

• To provide Comparison with Related Work

• To demonstrate Original Contribution

• To revisit Success Criteria

• To determine Limitations

• To predict Future Work

• To foresee Future Academic & Industrial Impact
7.1 Thesis Summary

In this PhD thesis, a formal and compositional framework for the development of monitoring systems for parallel computer systems is introduced. The proposed approach uses a single formalism, namely, Interval Temporal Logic (ITL), for specifications of and reasoning about correctness properties. This approach uses an executable subset of ITL, namely, Tempura, to monitor system behaviour at runtime and build correctness properties in order to deliver a property check against system behaviour via the runtime verifier AnaTempura.

The proposed approach, Parallel Runtime Verification Framework (PRVF), is intended to monitor parallel system architectures to ensure correctness properties. I consider in this approach concurrency forms, communication models, and execution modes of parallel systems under scrutiny. Additionally, this approach considers models for the management of resource allocation, delay and timeout agents to increase the robustness and reliability of the proposed framework in such cases.

The implementation of the proposed framework shows its comprehensibility and ability to model generic parallel systems architectures. The models of concurrency, communication, execution models enable it to deal realistically with varieties of parallel system architectures such as multi-core processor architecture, and Java Remote Method Invocation (RMI). The proposed framework allows systems to run either true concurrency or interleaving concurrency forms. Also, it allows systems to communicate via either shared-variable or message-passing models. The framework takes into consideration different execution modes such as Synchronous and Asynchronous. Assertion points mechanism allows systems to run globally or locally to exchange assertion data in order to check the desired properties at runtime to satisfy correctness criteria of the whole system. It also models resource allocation to manage the process of acquisition of shared resources and make them accessible systematically. Delay in the execution of a certain global resource within a given time causes a termination of the system holding the global
A benchmark case study of cache controller was implemented to demonstrate robustness of the proposed framework. The cache controller is composed of three cores (processors), each core is intended to execute read/write operations to/from memory addresses which are requested to be made upon private L2 cache memory. The design of the cache controller system considers a realistic hardware design as modelled in hardware description language SystemVerilog [213]. The coherence protocol, namely, MSI is modelled and implemented to deliver consistency property of the cache memory and main memory. For simplification sake, I did not use a bus in the cache controller system. Alternatively, the snoopy cache coherence protocol is implemented to ensure that two processors that attempt to write to the same block at the same time are strictly ordered serially and atomically. This situation is called data race where only one processor wins the write operation. The concurrency form used for the cache controller case study is true concurrency which implies the application of mutual exclusion and subsequently lock-based solution to enforce the synchronisation property of the cache memory. However, the lock-based solution was not used because the coherence protocol MSI was used instead. In other words, such a protocol avoids the use of locks, which leads to delay and sometimes concurrency bugs such as deadlock and then termination of the execution. The MSI protocol has a mechanism to mark the modified shared resource, for instance, the cache block, as dirty cache block in case it has been modified and is inconsistent with other cache blocks or the corresponding main memory address. This mechanism allows the processor to continue its computation and avoid waiting until the shared resource gets unlocked which saves time and delivers consistency property.

The proposed model was randomly and independently evaluated using an external tool for this purpose in order to make the judgement upon the model unbiased. The external tool is MATLAB and for this sake, AnaTempura has been fully integrated with MATLAB. AnaTempura and MATLAB make a strongly homogeneous pair because they complements each other. Discussion, related work, and comparative analysis were presented in this PhD thesis.
7.2 Comparison with Related Work

A memory model of the proposed approach using a well-defined formalism Interval Temporal Logic (ITL) plays a major role in runtime verification of parallel programs because parallel programs need to systematically manage their access and use of shared resources in order to deliver consistent memory model and consequently global correctness properties satisfaction. In the related work in Chapter 2, Framing Variable and Transactional Memory (TM) approaches have been discussed. These approaches tackle a drawback of the formalism framework of Interval Temporal Logic (ITL) which is the absence of memory model. Interval Temporal Logic (ITL) has two kinds of variables which are static variable and state variable. The static variable does not change over time, whereas the state variable gets changed over time. The state variable has the flexibility to get updated in different states or over intervals, but the problem is that the state variable’s value does not get inherited to the next states or over intervals; it becomes undefined.

Framing Variable is initiated by Hale [107] to overcome this shortcoming of the design of Interval Temporal Logic (ITL) formalism framework [190]. Also, Duan [295, 78] investigated Framing Variable and subsequently Projection Temporal Logic (PTL) is introduced as an ITL extension. Moreover, Duan introduced a new executable subset of PTL, namely, Framed Tempura. Framed Tempura has new operators such as projection \( \text{prj} \), synchronous communication \( \text{await} \), and framing operator \( \text{frame} \). However, framing variable has what is called framing problem where an explicit statement has to be made if a variable does not change, bearing in mind that the memory cell update is a very costly operation.

Alternatively, a \textit{stable} operator is used to model a memory which is intended to stabilise a list at different states or over intervals. A list construction is more powerful than a state variable construction intended to be framed to model benchmarks memory model such as the case study, cache controller.

On the other hand, El-kustaban [82, 80] formalised Transactional Memory (TM) in Interval
CHAPTER 7. CONCLUSION

Temporal Logic (ITL). However, there are still aspects that need to be imported to the provable abstract TM such as nested transactions and mechanisms of updating memory. The application of TM is limited and the debugging is difficult to place a breakpoint within the transaction. Transactional Memory (TM) has two major drawbacks which are space overhead and latency [68]. TM requires significant amounts of global and per-thread meta-data. Also, TM has high single-thread latency, usually two times compared to the lock-based technique.

The proposed approach adopts the lock-based technique although it is not been demonstrated in the case study of cache controller due to the involvement of cache coherence MSI Protocol. MSI protocol leaves a marker on modified shared cache block to indicate its state of coherence, to show whether it is consistent or inconsistent. Mutual Exclusion (Mutex) uses the lock-based technique; although it limits concurrency, it offers single-thread latency. Whereas, Transactional Memory (TM) has higher latency, it scales well [68].

7.3 Original Contribution

This PhD thesis develops a unified formal framework for the specification, verification, and implementation of Parallel Runtime Verification Framework (PRVF) using a single well-defined formalism, namely, Interval Temporal Logic (ITL). The proposed framework achieves:

- A general computational model for parallel computing architectures such as Multi-core, Java RMI. The framework fits any parallel computing architectures due to its comprehensibility and flexibility. It can be tailored according to the architecture design patterns.

- An executable version of the abstraction level of systems being implemented using Parallel Runtime Verification Framework (PRVF). A high-level (abstract) specification of a case study of cache controller is implemented using the framework PRVF in ITL. A low-level (concrete) design and implementation of the cache controller in Tempura/AnaTempura are delivered.
CHAPTER 7. CONCLUSION

• A formal executable specification of the cache controller system associated with the cache coherence protocol (MSI) is delivered in addition to snoopy protocol. A formal specification and verification of complete realistic behaviour of processor, cache memory, main memory. A formal modelling and concrete implementation of correctness properties for the cache controller system.

• A general computational model for handling different concurrency forms, communication models, and execution modes of parallel computing systems. In addition to a formal model of resource allocation, delay and timeout agents.

• A general algorithmic description of (PRVF) in terms of handling all perspectives of parallelism mentioned above in addition to the delivery of local/global properties verification at local/global levels of the framework.

7.4 Success Criteria Revisited

In the introduction chapter, number of success criteria is set as a measurement for this research. These success criteria are revised at this stage of this research to make a judgement according to what has been met of these success criteria which are:

1. **Compositional requirements from several sources to handle local and global systems correctness:**

   This criterion allows to specify and reason about global systems correctness of several parallel programs. The development of our framework considers the composition of high-level abstract specifications of parallel programs in order to deliver the correctness properties of global systems. This new feature is the basis of performing further verification of low-level concrete design and implementation of such programs (see Chapters 4, 5, 6).

2. **Compositional collection of assertion data from several sources to handle True/Interleaving Concurrency associated with Shared-Variable approach:**

   178
Parallel programs run in either True/Interleaving concurrency associated with Shared-Variable communication model which have the ability to send and receive assertion data from several sources to verify local/global correctness properties (see Chapters 3, 4, 5, 6).

3. **Compositional collection of assertion data from several sources to handle Synchronous/Asynchronous Communication links Channels/Shunts associated with Message-Passing approach:**

Parallel programs running in either Synchronous/Asynchronous execution modes associated with Message-Passing communication model have the ability to send and receive assertion data from several sources to verify local/global correctness properties. Synchronous communication links use a construct, namely, *Channels*, while Asynchronous communication links use the construct, namely, *Shunts* (see Chapter 3).

4. **The ability to execute agents concurrently and the introduction of resource allocation agents, and Delay and Timeout agents to model delay and timeout behaviour:**

The management of resource allocation, delay and timeout agents play a major role in increasing the robustness and reliability of the framework in such cases. Agents running in parallel need to be coordinated when they access shared resources. Moreover, timing is modelled to increase performance of such monitoring systems and to avoid deadlock situation in case agents do not respond timely (see Chapter 3).

5. **The use of lock-based technique to enforce Mutual Exclusion to deliver synchronisation:**

The Shared-Variable based approach needs the synchronisation mechanism to deliver consistent and reliable resources which are shared by many parallel programs. Mutual Exclusion is applied via the use of lock-based solution upon such cases. The lock-based solution affects the performance of such programs but endorses the correctness of global properties
of parallel programs (see Chapters 3, 4).

6. **Checking correctness property of local systems at local/global levels.** The inference of the correctness of global property from the correctness of a set of local properties of global systems:

Local systems can check their correctness properties locally at the local level (local verification & assertion phase) concurrently. The Framework allows such systems to perform this kind of correctness properties locally. Once local verification are made, the global verification phase at the global level collects all local correctness properties to form a global correctness property out of the local ones (see Chapters 3, 5, 6).

### 7.5 Limitations

The proposed computational model, Parallel Runtime Verification Framework (PRVF), has the following limitations:

- The decomposition paradigm allows breaking down the complex large systems into small groups accordingly in order to manage and coordinate their computation. The application of decomposition paradigm in this approach helps to model the correctness property of global properties. Guidelines for the mechanism of the decomposition of global properties would be helpful to a systematic understanding of the construction of such correctness properties.

- The proposed framework can not deal with the verification of parallel programs that run on several hosts. Multiple hosts have different environments which might infer the correctness of such programs and consequently harden the verification task. Programs being run in different hosts are subjected to different assumptions and commitments about the environments of those hosts.
CHAPTER 7. CONCLUSION

• As a consequent of the previous limitation, the proposed model of the monitoring system in this research can only deal with programs that run on the same host. The assumptions and commitments about the environment of that host are identical; therefore, the verification task of the delivery of correctness properties is conveniently performed.

7.6 Future Work

Bob Floyd [93] and Tony Hoare [118] introduced pre- and post-conditions, what is so called Hoare triple or logic, to verify systems at this level of abstraction. System $S$ satisfies a specification formulated as predicate pair of precondition $P$ and postcondition $R$. Precondition $P$ states the assumptions made of system $S$ before the system gets executed, whereas, Postcondition $R$ states the commitment which should be met after the execution of the system.

$$\{ P \} S \{ R \}$$

As a consequence of Hoare’s Logic [118], the Assumption/Commitment style is developed to verify a set of properties of interest such as Cau and Collette in [45], Moszkowski in [195], and Zedan et al in [297]. Moszkowski [195] is the first to introduce Hoare’s logic to Interval Temporal Logic (ITL). Hoare’s logic’s clause can be expressed in ITL as follows:

$$\vdash \omega \land Sys \supset fin \omega'$$

where $\omega$ and $\omega'$ are state formulas have no temporal operators, $Sys$ is some arbitrary temporal formula, and $fin \omega'$ is true on an interval iff $\omega'$ is true in the interval’s final state. Moszkowski [195] addressed a drawback of pre- and post-condition approach which is the unsuitability for the specification and verification of continuous and parallel systems. Moszkowski claims the remedy of pre- and post-condition approach via the introduction of the Assumption/Commitment approach. According to Moszkowski [195], the first consideration of the latter approach is credited to Francez and Pnueli [95]. The expression of Assumption/Commitment in ITL is as follows:
CHAPTER 7. CONCLUSION

\[ \vdash \omega \land As \land Sys \supset Co \land fin\omega' \]

where:
\( \omega \): state formula about initial state,
\( As \): assumption about overall interval,
\( Sys \): the system under consideration,
\( Co \): commitment about overall interval,
\( \omega' \): state formula about final state.

According to Zhou [299], when compositional reasoning about systems run in parallel \( Sys_1 \) and \( Sys_2 \), the composition can be modelled in Assumption/Commitment style as follows:

\[ \vdash \omega_1 \land As_1 \land Sys_1 \supset Co_1 \land fin\omega'_1 \]
\[ \vdash \omega_2 \land As_2 \land Sys_2 \supset Co_2 \land fin\omega'_2 \]

\[ \vdash \omega \land (Sys_1 \parallel Sys_2) \supset Co \land fin\omega' \]

where:
\[ \vdash \omega \subseteq \omega_1 \land \omega_2 \]
\[ \vdash As \lor Co_1 \subseteq As_2 \]
\[ \vdash As \lor Co_2 \subseteq As_1 \]
\[ \vdash Co_1 \lor Co_2 \subseteq Co \]
\[ \vdash fin\omega'_1 \land fin\omega'_2 \subseteq fin\omega' \]

This kind of compositional reasoning about correctness properties enables monitoring system at a high-level (abstraction), such as PRVF, to deal with parallel programs running on several hosts. To transform these specifications expressed in Assumption/Commitment style into a low-level (implementation), an executable version is needed. Transforming these specifications into an executable version is intended in the future.
CHAPTER 7. CONCLUSION

The transformation of Assumption/Commitment from a high-level into a low-level shifts the proposed framework towards monitoring programs running in different hosts because the availability of modelling and execution of different environments might exist.

7.7 Future Impact

7.7.1 Academic

The development of runtime verification benchmarks that include parallel systems is a promising research topic due to the evolutionary shift in manufacturing multi-core architectures and the wide adoption of such architectures. Modern computers use multi-core processor architectures at the hardware/software levels in their design. Therefore, the performance and correctness of such applications are vital for daily life. The continuity of such research topic is commercially profitable and academically promising.

7.7.2 Industrial

The emergence of simulation based verification and validation techniques such as virtual commissioning is a sudden shift solution for testing automation systems even in the absence of the process that is subjected to control. The Distributed Control Systems (DCS) which are intended to control industrial processes might involve thousands of instruments, actuators and controllers running in parallel to boost performance and save time. These giant control systems are complex and rely on actuators and sensors. The probability of failures is high in harsh industrial environments due to the possibility of malfunctions and defects in actuators, sensors, or process equipment. According to OREDA, 92% of automated control and safety malfunctions of 10 international petroleum groups encountered are due to sensor or actuator malfunctions. This kind of defects of sensors and actuators implies the introduction of a failure-tolerant design to overcome this vulnerability of control systems. The shut down of any process in the field due to sensors or actuators failures might cause significant waste of materials and work hours which
lead to profitability reduction. The lost in annual revenue in the United States caused by sensors and actuators malfunctions alone is tens of billion dollars [230].

The development of runtime verification benchmarks that include parallel control systems consider failure-tolerant design to avoid shutting down processes in the field due to sensors or actuators failures which cause catastrophic loses in annual revenues of industrials. Savolainen et al. [230] propose a runtime verification framework using plant simulation models created during the plant design process. The verification technique used in [230] is able to cover control software errors, sensors and actuators errors. However, I believe the development of parallel runtime verification framework is vital due to concurrency forms, communication models, and execution modes perspectives. Their importance to deliver correctness properties for such control systems which involve thousands of instruments, sensors, actuators which eventually run in parallel, access shared resources, and executes differently.

Another industrial impact of the development of parallel runtime verification is the correctness property of hardware and software writing of parallel processing programs. As the number of cores is doubled every two years, programmers who are interested in increasing performance have to be parallel programmers. Manufacturing hardware and software for multi-core processor architectures that make the writing of correct parallel processing programs leads to efficiency in performance and power as the number of cores per chip scales geometrically. The development of parallel runtime verification framework for such industrial fields of multi-core systems is a sudden shift towards correctness and performance [213].
Bibliography


[78] Z. Duan. An extended interval temporal logic and a framing technique for temporal logic programming. 1996.


BIBLIOGRAPHY


BIBLIOGRAPHY


212


Appendix A

Appendix A: Simulations & Animation
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.1: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 0

Figure A.2: CACHE CONTROLLER SIMULATION IN ANA TEMPURA AT STATE 0

Figure A.3: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 0
Figure A.4: Cache Controller Execution in Tempura at State 1

Figure A.5: Cache Controller Simulation in AnaTempura at State 1

Figure A.6: Local States & Properties of Processors 0, 1, 2 at State 1
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.7: Cache Controller Execution in Tempura at State 2

Figure A.8: Cache Controller Simulation in AnaTempura at State 2

Figure A.9: Local States & Properties of Processors 0, 1, 2 at State 2
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.10: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 3

Figure A.11: CACHE CONTROLLER SIMULATION IN ANA TEMPURA AT STATE 3

Figure A.12: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 3
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.13: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 4

Figure A.14: CACHE CONTROLLER SIMULATION IN ANATEMPURA AT STATE 4

Figure A.15: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 4

221
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.16: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 5

Figure A.17: CACHE CONTROLLER SIMULATION IN ANA TEMPURA AT STATE 5

Figure A.18: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 5

222
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.19: Cache controller execution in Tempura at State 6

Figure A.20: Cache controller simulation in AnaTempura at State 6

Figure A.21: Local states & properties of processors 0, 1, 2 at State 6

223
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.22: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 7

Figure A.23: CACHE CONTROLLER SIMULATION IN ANA TEMPURA AT STATE 7

Figure A.24: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 7
APPENDIX A. APPENDIX A: SIMULATIONS & ANIMATION

Figure A.25: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 8

Figure A.26: CACHE CONTROLLER SIMULATION IN ANATEMPURA AT STATE 8

Figure A.27: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 8
Figure A.28: CACHE CONTROLLER EXECUTION IN TEMPURA AT STATE 9

Figure A.29: CACHE CONTROLLER SIMULATION IN ANATEMPURA AT STATE 9

Figure A.30: LOCAL STATES & PROPERTIES OF PROCESSORS 0, 1, 2 AT STATE 9
Appendix B

Appendix B: Tempura Code for Cache Controller
/* Mode: C */

L2_Cache_MSI_V1_5.t

This file is part of Tempura: Interval Temporal Logic interpreter.

Copyright (C) 1998-2017 Nayef H. Alshammari, Antonio Cau

Tempura is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

Tempura is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with Tempura. If not, see <http://www.gnu.org/licenses/>.

define nprocessors = 3.
define nmemorylocations = 16.
define ncachelocations = 8.
define nlocations = 8. /* for Tag and Index */
define initial_value = -8.
define initial_value2 = -16.
define modified = 0.
define shared = 1.
define invalid = 2.

load "../library/conversion".
load "../library/exprog".
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```tcl
38 load "../library/tcl".
39
40 /* anatempura 0 */
41 /* anatempura 1 */
42 /* anatempura 2 */
43
44 /* tcl L2_Cache_MSI_v3 3*/
45
46
47 set print_states=true.
48 set break_is_abort=false.
49
50 define avar1(X,a) = {
51    X[a]
52 }.
53
54 define aval1(X,b) = {
55    X[b]
56 }.
57
58 define atime1(X,c) = {
59    strint(X[c])
60 }.
61
62 define atime_micro1(X,d) = {
63    strint(X[d])
64 }.
65
66 define prog_send2(A,X) = {
67    format("!E: prog%s %s
",ctype(A),parstr([X]))
68 }.
69
70 define prog_send_ne1(A,X) = {
71    empty and format("!E: prog%s %s
",ctype(A),parstr([X]))
72 }.
73
74 /* 2^0 to 2^127 */
75 define bits=[1,2,4,8,16,32,64,128,256,512,1024,2048,
229```
4096, 8192, 16384, 32768, 65536, 131072, 262144,
524288, 1048576, 2097152, 4194304, 8388608,
16777216, 33554432, 67108864, 134217728,
268435456, 536870912, 1073741824, 2147483648].

\[
\text{define bit (Bitno, Number) = \{ ((Number div bits[Bitno]) mod 2 = 1)\}.
\]

\[
\text{define inttobitslist_msb(X) = \{
[bit(31,X), bit(30,X), bit(29,X), bit(28,X),
bit(27,X), bit(26,X), bit(25,X), bit(24,X),
bit(23,X), bit(22,X), bit(21,X), bit(20,X),
bit(19,X), bit(18,X), bit(17,X), bit(16,X),
bit(15,X), bit(14,X), bit(13,X), bit(12,X),
bit(11,X), bit(10,X), bit(9,X), bit(8,X),
bit(7,X), bit(6,X), bit(5,X), bit(4,X),
bit(3,X), bit(2,X), bit(1,X), bit(0,X)]
\}.}
\]

\[
\text{define single_bit(X) = \{
exists i: \{
\text{skip and for (i<32) do }\{
(if bit(i,X) then "1" else "0")
\}
\}
\}.
\]

\[
\text{define msb32_2(Y,X) = \{
(if bit(Y,X) then "1" else "0")
\}.}
\]

\[
\text{define msb_16_index(X) ={\}
(if bit(3,X) then "1" else "0") +
(if bit(2,X) then "1" else "0") +
(if bit(1,X) then "1" else "0") +
(if bit(0,X) then "1" else "0")
\}.
\]

\[
\text{define msb_8_index(X) = \{}
\}
\]

230
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

116  (if bit(2,X) then "1" else "0") +
117  (if bit(1,X) then "1" else "0") +
118  (if bit(0,X) then "1" else "0")
119  ).
120  define msb_14_index(X) = {
121  (if bit(13,X) then "1" else "0") +
122  (if bit(12,X) then "1" else "0") +
123  (if bit(11,X) then "1" else "0") +
124  (if bit(10,X) then "1" else "0") +
125  (if bit(9,X) then "1" else "0") +
126  (if bit(8,X) then "1" else "0") +
127  (if bit(7,X) then "1" else "0") +
128  (if bit(6,X) then "1" else "0") +
129  (if bit(5,X) then "1" else "0") +
130  (if bit(4,X) then "1" else "0") +
131  (if bit(3,X) then "1" else "0") +
132  (if bit(2,X) then "1" else "0") +
133  (if bit(1,X) then "1" else "0") +
134  (if bit(0,X) then "1" else "0")
135  }.
136  define msb_7bits_addr(X) = {
137  (if bit(6,X) then "1" else "0") +
138  (if bit(5,X) then "1" else "0") +
139  (if bit(4,X) then "1" else "0") +
140  (if bit(3,X) then "1" else "0") +
141  (if bit(2,X) then "1" else "0") +
142  (if bit(1,X) then "1" else "0") +
143  (if bit(0,X) then "1" else "0")
144  }.
145  define msb(X) = {
146  (if bit(31,X) then "1" else "0") +
147  (if bit(30,X) then "1" else "0") +
148  (if bit(29,X) then "1" else "0") +
149  (if bit(28,X) then "1" else "0") +
150  (if bit(27,X) then "1" else "0") +
151  (if bit(26,X) then "1" else "0") +
152  (if bit(25,X) then "1" else "0") +
153  (if bit(24,X) then "1" else "0") +
154  (if bit(23,X) then "1" else "0") +

231
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

155 (if bit(22, X) then "1" else "0") +
156 (if bit(21, X) then "1" else "0") +
157 (if bit(20, X) then "1" else "0") +
158 (if bit(19, X) then "1" else "0") +
159 (if bit(18, X) then "1" else "0") +
160 (if bit(17, X) then "1" else "0") +
161 (if bit(16, X) then "1" else "0") +
162 (if bit(15, X) then "1" else "0") +
163 (if bit(14, X) then "1" else "0") +
164 (if bit(13, X) then "1" else "0") +
165 (if bit(12, X) then "1" else "0") +
166 (if bit(11, X) then "1" else "0") +
167 (if bit(10, X) then "1" else "0") +
168 (if bit(9, X) then "1" else "0") +
169 (if bit(8, X) then "1" else "0") +
170 (if bit(7, X) then "1" else "0") +
171 (if bit(6, X) then "1" else "0") +
172 (if bit(5, X) then "1" else "0") +
173 (if bit(4, X) then "1" else "0") +
174 (if bit(3, X) then "1" else "0") +
175 (if bit(2, X) then "1" else "0") +
176 (if bit(1, X) then "1" else "0") +
177 (if bit(0, X) then "1" else "0") +
178 }
179
180 define tag_field_cache(X) = {
181 (if bit(31, X) then "1" else "0") +
182 (if bit(30, X) then "1" else "0") +
183 (if bit(29, X) then "1" else "0") +
184 (if bit(28, X) then "1" else "0") +
185 (if bit(27, X) then "1" else "0") +
186 (if bit(26, X) then "1" else "0") +
187 (if bit(25, X) then "1" else "0") +
188 (if bit(24, X) then "1" else "0") +
189 (if bit(23, X) then "1" else "0") +
190 (if bit(22, X) then "1" else "0") +
191 (if bit(21, X) then "1" else "0") +
192 (if bit(20, X) then "1" else "0") +
193 (if bit(19, X) then "1" else "0") +
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```
(define index_field_cache_8(X) = {
    (if bit(2,X) then "1" else "0") +
    (if bit(1,X) then "1" else "0") +
    (if bit(0,X) then "1" else "0")
}).

(define_index_field_cache(X) = {
    (if bit(9,X) then "1" else "0") +
    (if bit(8,X) then "1" else "0") +
    (if bit(7,X) then "1" else "0") +
    (if bit(6,X) then "1" else "0") +
    (if bit(5,X) then "1" else "0") +
    (if bit(4,X) then "1" else "0") +
    (if bit(3,X) then "1" else "0") +
    (if bit(2,X) then "1" else "0") +
    (if bit(1,X) then "1" else "0") +
    (if bit(0,X) then "1" else "0")
}).

(define tag_field_memory(X) = {
    (if bit(31,X) then "1" else "0") +
    (if bit(30,X) then "1" else "0") +
    (if bit(29,X) then "1" else "0") +
    (if bit(28,X) then "1" else "0") +
    (if bit(27,X) then "1" else "0") +
    (if bit(26,X) then "1" else "0") +
    (if bit(25,X) then "1" else "0") +
    (if bit(24,X) then "1" else "0") +
    (if bit(23,X) then "1" else "0") +
    (if bit(22,X) then "1" else "0") +
    (if bit(21,X) then "1" else "0") +
    (if bit(20,X) then "1" else "0") +
    (if bit(19,X) then "1" else "0") +
    (if bit(18,X) then "1" else "0") +
    (if bit(17,X) then "1" else "0") +
    (if bit(16,X) then "1" else "0") +
    (if bit(15,X) then "1" else "0") +
    (if bit(14,X) then "1" else "0") +
    (if bit(13,X) then "1" else "0") +
    (if bit(12,X) then "1" else "0") +
    (if bit(11,X) then "1" else "0") +
    (if bit(10,X) then "1" else "0") +
    (if bit(9,X) then "1" else "0") +
    (if bit(8,X) then "1" else "0") +
    (if bit(7,X) then "1" else "0") +
    (if bit(6,X) then "1" else "0") +
    (if bit(5,X) then "1" else "0") +
    (if bit(4,X) then "1" else "0") +
    (if bit(3,X) then "1" else "0") +
    (if bit(2,X) then "1" else "0") +
    (if bit(1,X) then "1" else "0") +
    (if bit(0,X) then "1" else "0")
}).
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
(if bit(30, X) then "1" else "0") +
(if bit(29, X) then "1" else "0") +
(if bit(28, X) then "1" else "0") +
(if bit(27, X) then "1" else "0") +
(if bit(26, X) then "1" else "0") +
(if bit(25, X) then "1" else "0") +
(if bit(24, X) then "1" else "0") +
(if bit(23, X) then "1" else "0") +
(if bit(22, X) then "1" else "0") +
(if bit(21, X) then "1" else "0") +
(if bit(20, X) then "1" else "0") +
(if bit(19, X) then "1" else "0") +
(if bit(18, X) then "1" else "0") +
(if bit(17, X) then "1" else "0") +
(if bit(16, X) then "1" else "0") +
(if bit(15, X) then "1" else "0") +
(if bit(14, X) then "1" else "0") +
(if bit(13, X) then "1" else "0") +
(if bit(12, X) then "1" else "0") +
(if bit(11, X) then "1" else "0") +
}

define index_field_memory_16(X) = {
(if bit(3, X) then "1" else "0") +
(if bit(2, X) then "1" else "0") +
(if bit(1, X) then "1" else "0") +
(if bit(0, X) then "1" else "0") +
}

define index_field_memory(X) = {
(if bit(10, X) then "1" else "0") +
(if bit(9, X) then "1" else "0") +
(if bit(8, X) then "1" else "0") +
(if bit(7, X) then "1" else "0") +
(if bit(6, X) then "1" else "0") +
(if bit(5, X) then "1" else "0") +
(if bit(4, X) then "1" else "0") +
(if bit(3, X) then "1" else "0") +
}
```
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```cpp
272 (if bit(2, X) then "1" else "0") +
273 (if bit(1, X) then "1" else "0") +
274 (if bit(0, X) then "1" else "0")
}

276 define data_field(X) = {
277 (if bit(31, X) then "1" else "0") +
278 (if bit(30, X) then "1" else "0") +
279 (if bit(29, X) then "1" else "0") +
280 (if bit(28, X) then "1" else "0") +
281 (if bit(27, X) then "1" else "0") +
282 (if bit(26, X) then "1" else "0") +
283 (if bit(25, X) then "1" else "0") +
284 (if bit(24, X) then "1" else "0") +
285 (if bit(23, X) then "1" else "0") +
286 (if bit(22, X) then "1" else "0") +
287 (if bit(21, X) then "1" else "0") +
288 (if bit(20, X) then "1" else "0") +
289 (if bit(19, X) then "1" else "0") +
290 (if bit(18, X) then "1" else "0") +
291 (if bit(17, X) then "1" else "0") +
292 (if bit(16, X) then "1" else "0") +
293 (if bit(15, X) then "1" else "0") +
294 (if bit(14, X) then "1" else "0") +
295 (if bit(13, X) then "1" else "0") +
296 (if bit(12, X) then "1" else "0") +
297 (if bit(11, X) then "1" else "0") +
298 (if bit(10, X) then "1" else "0") +
299 (if bit(9, X) then "1" else "0") +
300 (if bit(8, X) then "1" else "0") +
301 (if bit(7, X) then "1" else "0") +
302 (if bit(6, X) then "1" else "0") +
303 (if bit(5, X) then "1" else "0") +
304 (if bit(4, X) then "1" else "0") +
305 (if bit(3, X) then "1" else "0") +
306 (if bit(2, X) then "1" else "0") +
307 (if bit(1, X) then "1" else "0") +
308 (if bit(0, X) then "1" else "0") +
309 (if bit(0, X) then "1" else "0")
310 }
```
define update_msi(i,B,L2CacheState,v) =
{
(forall j<ncachelocations :
  if j=B then { L2CacheState[i][j]:=v }
  else { stable(L2CacheState[i][j]) }
) }

define cs_text(V) = {
  if V=modified then "Modified"
  else if V=shared then "Shared"
  else if V=invalid then "Invalid"
  else "Error"
}.

define cpu_request(MainMemory,L2CacheMemory,L2CacheTag,L2CacheState,
Vbit,Dbit,x,RW,ADDR,DATA,j,Tick) = {
exists y,z,indexc, indexm, tag, datam, csx, csy, csz, tagx, tagy, tags, datax, datay, dataz,
stringx, stringy, strings, tmpwb, vbitx,vbity,vbitz, nsvbitx,nsvbity,nsvbitz, ncsx, ncsy, ncsz,
dbitx,dbity,dbitz,ndbitx,ndbity,ndbitz,ntagx,ntagy,ntagz,Sx,Sy,Sz,S,
cmx, cmy, cmz, ncmx, ncmy, ncmz, mm, nmm :
  y = (x+1) mod nprocessors and
  z = (x+2) mod nprocessors and
  indexc = ADDR mod ncachelocations and
  indexm = ADDR mod nmemorylocations and
  tag = ADDR div ncachelocations and
  csx = L2CacheState[x][indexc] and
  csy = L2CacheState[y][indexc] and
  csz = L2CacheState[z][indexc] and
  ncsx = next(L2CacheState[x][indexc]) and
  ncsy = next(L2CacheState[y][indexc]) and
  ncsz = next(L2CacheState[z][indexc]) and
  tagx = L2CacheTag[x][indexc] and
  tagy = L2CacheTag[y][indexc] and
  tagz = L2CacheTag[z][indexc] and
  ntagx = next(L2CacheTag[x][indexc]) and
  ntagy = next(L2CacheTag[y][indexc]) and
  ntagz = next(L2CacheTag[z][indexc]) and
}.
skip and

/*stringy = " " and*/
/*stringz = " " and*/
/*format("Processor %t, Cache is %t \n",x,cs_text(csx)) and*/
format("Processor %t, Cache is %t \n",z,cs_text(csz)) and
format("Processor %t, Cache is %t \n",y,cs_text(csy)) and

vbitx = {if csx = shared or csx = modified then 1 else 0} and
vbity = {if csy = shared or csy = modified then 1 else 0} and
vbitz = {if csz = shared or csz = modified then 1 else 0} and
nvbitx = {if ncsx = shared or ncsx = modified then 1 else 0} and
nvbity = {if ncsy = shared or ncsy = modified then 1 else 0} and
nvbitz = {if ncsz = shared or ncsz = modified then 1 else 0} and

dbitx = {if csx = modified then 1 else 0} and
dbity = {if csy = modified then 1 else 0} and
dbitz = {if csz = modified then 1 else 0} and
ndbitx = {if ncsx = modified then 1 else 0} and
ndbity = {if ncsy = modified then 1 else 0} and
ndbitz = {if ncsz = modified then 1 else 0} and

if RW = 0 then { /* read */
stringx = {if tag = tagx and (csx = shared or csx = modified) then "Read Hit" else "Read ...
Miss") and
stringy = {if tag = tagy and (csy = shared or csy = modified) then "Read Hit" else "Read ...
Miss") and
stringz = {if tag = tagz and (csz = shared or csz = modified) then "Read Hit" else "Read ...
Miss") and

if tag = tagx then {
/* read hit cache x */
if csx = shared or csx = modified then {
/* normal hit */
/*/stringx = "Read hit" and*/
format("State %d: Processor %t getting data from Cache[%t]\n", j, x, indexc) and
memory_unchanged(MainMemory) and
DATA = L2CacheMemory[x][indexc] and
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, x) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and

/*stringy = " " and*/
/*stringz = " " and*/
update_msi(x, indexc, L2CacheState, csx) and
update_msi(y, indexc, L2CacheState, csy) and
update_msi(z, indexc, L2CacheState, csz)

} else { /* cache line x invalid */
/*stringx = "Read miss" and*/
if csy = invalid and csz = invalid then {
format("State \%d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
/*stringy = "Read miss" and*/
/*stringz = "Read miss" and*/
memory_unchanged(MainMemory) and
update_msi(x, indexc, L2CacheState, shared) and
update_msi(y, indexc, L2CacheState, invalid) and
update_msi(z, indexc, L2CacheState, invalid)
} else if csy = invalid and csz = modified then {
if tag = tagz then { /* read hit in cache z */
/*stringz = "Read hit" and*/
/*stringy = "Read miss" and*/
format("State \%d: Getting data from Cache of processor \%t \n", j, z) and
DATA = L2CacheMemory[z][indexc] and
write_to_memory(MainMemory, x, indexm, DATA, Tick) and
format("State \%d: Coherence step for processor \%t\n", j, z) and
write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
update_msi(x, indexc, L2CacheState, shared) and
update_msi(y, indexc, L2CacheState, invalid) and
update_msi(z, indexc, L2CacheState, shared)
} else { /* read miss in cache z */
/*stringz = "Read miss" and*/
/*stringy = "Read miss" and*/
```
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

425  format("State %d: Getting data from global memory\n", j) and
426  tmpwb = 8*tagz+indexc and
427  dataz = L2CacheMemory[x][indexc] and
428  format("State %d: Processor %t, write-back cache[%t] with tag %t and data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
429  DATA = MainMemory[indexm] and
430  write_to_memory(MainMemory,z, tmpwb, dataz,Tick) and
431  write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
432  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
433  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
434  update_msi(x,indexc,L2CacheState,shared) and
435  update_msi(y,indexc,L2CacheState,invalid) and
436  update_msi(z,indexc,L2CacheState,invalid)
437  }
438  } else if csy = modified and csz = invalid then {
439  if tag = tagy then { /* read hit in cache y */
440  /*stringy = "Read hit" and*/
441  /*stringz = "Read miss" and*/
442  format("State %d: Getting data from Cache of processor %t \n", j, y) and
443  DATA = L2CacheMemory[y][indexc] and
444  write_to_memory(MainMemory,x,indexm,DATA,Tick) and
445  format("State %d: Coherence step for processor %t\n", j, y) and
446  write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
447  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
448  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
449  update_msi(x,indexc,L2CacheState,shared) and
450  update_msi(z,indexc,L2CacheState,invalid) and
451  update_msi(y,indexc,L2CacheState,shared)
452  } else { /* read miss in cache y */
453  /*stringy = "Read miss" and*/
454  /*stringz = "Read miss" and*/
455  format("State %d: Getting data from global memory\n", j) and
456  tmpwb = 8*tagy+indexc and
457  datay = L2CacheMemory[y][indexc] and
458  format("State %d: Processor %t, write-back cache[%t] with tag %t and data %t to memory[%t] \n", j, y, indexc, tagy, datay, tmpwb) and
459  DATA = MainMemory[indexm] and
460  write_to_memory(MainMemory,y, tmpwb, datay,Tick) and
461  write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and

239
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and

update_msi(x,indexc,L2CacheState,shared) and

update_msi(y,indexc,L2CacheState,invalid) and

update_msi(z,indexc,L2CacheState,invalid)
}

} else if csy = shared then {

if tag = tagy then { /* read hit in cache y */
  /*stringy = "Read hit" and*/
  /*stringz = " " and*/
  format("State %d: Getting data from Cache of processor %t \n", j, y) and
  DATA = L2CacheMemory[y][indexc] and
  memory_unchanged(MainMemory) and
  write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
  update_msi(x,indexc,L2CacheState,shared) and
  update_msi(y,indexc,L2CacheState,shared) and
  update_msi(z,indexc,L2CacheState,csz) and

                      } else { /* read miss in cache y */
  /*stringy = "Read miss" and*/
  if csz = shared then {
  if tag = tagz then { /* read hit in cache z */
   /*stringz = "Read hit " and*/
   format("State %d: Getting data from Cache of processor %t \n", j, z) and
   DATA = L2CacheMemory[z][indexc] and
   memory_unchanged(MainMemory) and
   write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
   cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
   cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
   update_msi(x,indexc,L2CacheState,shared) and
   update_msi(y,indexc,L2CacheState,invalid) and
   update_msi(z,indexc,L2CacheState,shared)
                       ) else { /* read miss in cache z */
  /*stringz = "Read miss" and*/
  format("State %d: Getting data from global memory\n", j) and
  DATA = MainMemory[indexm] and
  write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and

APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

503  cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
504  memory_unchanged(MainMemory) and
505  update_msi(x,indexc,L2CacheState,shared) and
506  update_msi(y,indexc,L2CacheState,invalid) and
507  update_msi(z,indexc,L2CacheState,invalid)
508  }
509  } else {
510    /*stringz = "Read miss" and*/
511    format("State %d: Getting data from global memory\n", j) and
512    DATA = MainMemory[indexm] and
513    write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
514    cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
515    cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
516    memory_unchanged(MainMemory) and
517    update_msi(x,indexc,L2CacheState,shared) and
518    update_msi(y,indexc,L2CacheState,invalid) and
519    update_msi(z,indexc,L2CacheState,invalid)
520  }
521  }
522  } else {
523    if csz = shared then {
524      if tag = tagz then { /* read hit in cache z */
525        /*stringz = "Read hit" and*/
526        /*stringy = " " and*/
527        format("State %d: Getting data from Cache of processor %t \n", j, z) and
528        DATA = L2CacheMemory[z][indexc] and
529        memory_unchanged(MainMemory) and
530        write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
531        cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
532        cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
533        update_msi(x,indexc,L2CacheState,shared) and
534        update_msi(y,indexc,L2CacheState,invalid) and
535        update_msi(z,indexc,L2CacheState,shared)
536      } else { /* read miss in cache z */
537        /*stringz = "Read miss" and*/
538        /*stringy = "Read miss" and*/
539        format("State %d: Getting data from global memory\n", j) and
540        DATA = MainMemory[indexm] and
541        write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
542      } else { /* read miss in cache z */
543      /*stringz = "Read miss" and*/
544      /*stringy = "Read miss" and*/
545      format("State %d: Getting data from global memory\n", j) and
546      DATA = MainMemory[indexm] and
547      write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
241
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
542 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
543 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
544 memory_unchanged(MainMemory) and
545 update_msi(x, indexc, L2CacheState, shared) and
546 update_msi(y, indexc, L2CacheState, invalid) and
547 update_msi(z, indexc, L2CacheState, invalid)
548 }
549 }
550 } else { /*stringz = "Read miss" and*/
551 /*stringy = "Read miss" and*/
552 format("State %d: Getting data from global memory\n", j) and
553 DATA = MainMemory[indexm] and
554 write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
555 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
556 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
557 memory_unchanged(MainMemory) and
558 update_msi(x, indexc, L2CacheState, shared) and
559 update_msi(y, indexc, L2CacheState, invalid) and
560 update_msi(z, indexc, L2CacheState, invalid)
561 }
562 }
563 }
564 } else { /* read miss tag \neq tagx */
565 /*stringx = "Read miss" and*/
566 if csx = invalid then { /* cache line x is invalid */
567 if csy = invalid and csz = invalid then {
568 /*stringz = "Read miss" and*/
569 /*stringy = "Read miss" and*/
570 format("State \%d: Getting data from global memory\n", j) and
571 DATA = MainMemory[indexm] and
572 write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
573 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
574 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
575 memory_unchanged(MainMemory) and
576 update_msi(x, indexc, L2CacheState, shared) and
577 update_msi(y, indexc, L2CacheState, invalid) and
578 update_msi(z, indexc, L2CacheState, invalid)
579 } else if csy = invalid and csz = modified then {
580 if tag = tagz then { /* read Hit in cache z */
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
/*stringz = "Read hit" and*/
/*stringy = " " and*/
format("State %d: Getting data from Cache of processor %t \n", j, z) and
DATA = L2CacheMemory[z][indexc] and
write_to_memory(MainMemory,x,indexm,DATA,Tick) and
format("State %d: Coherence step for processor %t\n", j, z) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,shared)
} else { /* read miss in cache z */
/*stringz = "Read miss" and*/
/*stringy = "Read miss" and*/
format("State %d: Getting data from global memory\n", j) and
tmpwb = 8*tagz+indexc and
dataz = L2CacheMemory[z][indexc] and
format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
DATA = MainMemory[indexm] and
write_to_memory(MainMemory,z,tmpwb,dataz,Tick) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,invalid)
}
else if csy = modified and csz = invalid then {
if tag = tagy then { /* read Hit in cache y */
/*stringy = "Read hit" and*/
/*stringz = "Read miss" and*/
format("State %d: Getting data from Cache of processor %t \n", j, y) and
DATA = L2CacheMemory[y][indexc] and
format("State %d: Coherence step for processor %t\n", j, y) and
write_to_memory(MainMemory,x,indexm,DATA,Tick) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
```
243
APPENDIX B: APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
update_msi(x, indexc, L2CacheState, shared) and
update_msi(z, indexc, L2CacheState, invalid) and
update_msi(y, indexc, L2CacheState, shared)
} else { /* read Miss in cache y */
/*stringy = "Read miss" and*/
/*stringz = "Read miss" and*/
format("State %d: Getting data from global memory\n", j) and
tmpwb = 8*tagy+indexc and
datay = L2CacheMemory[y][indexc] and
format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t]\n", j, y, indexc, tagy, datay, tmpwb) and
DATA = MainMemory[indexm] and
write_to_memory(MainMemory, y, tmpwb, datay, Tick) and
write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
update_msi(x, indexc, L2CacheState, shared) and
update_msi(y, indexc, L2CacheState, invalid) and
update_msi(z, indexc, L2CacheState, invalid)
}
} else if csy = shared then {
if tag = tagy then { /* read hit in cache y */
/*stringy = "Read hit" and*/
/*stringz = "" and*/
format("State %d: Getting data from Cache of processor %t\n", j, y) and
DATA = L2CacheMemory[y][indexc] and
memory_unchanged(MainMemory) and
write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z) and
update_msi(x, indexc, L2CacheState, shared) and
update_msi(z, indexc, L2CacheState, csz) and
update_msi(y, indexc, L2CacheState, shared)
} else { /* read miss in cache y */
/*stringy = "Read miss" and*/
if csz = shared then {
if tag = tagz then { /* read hit in cache z */
/*stringz = "Read hit" and*/
```
format("State %d: Getting data from Cache of processor \%t \n", j, y) and
DATA = L2CacheMemory[z][indexc] and
memory_unchanged(MainMemory) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,shared)
} else { /* read miss in cache z */
/*stringz = "Read miss" and*/
format("State %d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,invalid)
} else {
/*stringz = "Read miss" and*/
format("State %d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,invalid)
}
}
} else {
/*stringz = "Read miss" and*/
format("State %d: Getting data from cache %z", j) and
/*stringy = " ", x = tagz */
if tagz = shared then {
/*stringy = " ", x = tagz */
if tag = tagz then { /* read hit in cache %z */
/*stringy = "Read hit" and*/
/*stringy = " " and*/}
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

format("State \%d: Getting data from Cache of processor \%t \n", j, z) and
DATA = L2CacheMemory[z][indexc] and
memory_unchanged(MainMemory) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,shared)
} else /* read miss in cache z */
/*stringz = "Read miss" and*/
/*stringy = "Read miss" and*/
format("State \%d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,invalid)
}
} else {
/*stringy = "Read miss" and*/
/*stringz = "Read miss" and*/
format("State \%d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,invalid) and
update_msi(z,indexc,L2CacheState,invalid)
}
} else if csx = shared then /* cache line x is shared */
if csz = shared and tagz = tag then /* cache line z is shared and
a read hit on cache line z */
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
/*stringz = "Read hit" and*/
/*stringy = " " and*/
format("State %d: Getting data from Cache of processor %t \n", j, z) and
DATA = L2CacheMemory[z][indexc] and
memory_unchanged(MainMemory) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,csy) and
update_msi(z,indexc,L2CacheState,shared)
} else { /* cache line z is not shared or a read miss on cache line z */
    /*if tagz != tag then { stringz = "Read miss" } else { stringz = " " } and*/
    if csy = shared and tagy = tag then { /* cache line y is shared and
        a read hit on cache line y */
      /*stringy = "Read hit" and*/
      format("State %d: Getting data from global memory\n", j) and
      DATA = MainMemory[indexm] and
      write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
      cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
      cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
      memory_unchanged(MainMemory) and
      update_msi(x,indexc,L2CacheState,shared) and
      update_msi(y,indexc,L2CacheState,csy) and
      update_msi(z,indexc,L2CacheState,csz)
    }
  } else { /* cache line x is modified, write-back */
```

247
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
/*stringy = " " and*/
/*stringz = " " and*/
format("State %d: Getting data from global memory\n", j) and
DATA = MainMemory[indexm] and
tmpwb = 8*tagx+indexc and
datax = L2CacheMemory[x][indexc] and
format("State %d: Processor %t, write-back cache[%t] with tag %t
and data %t to memory[%t] \n", j, x, indexc, tagx, datax, tmpwb) and
write_to_memory(MainMemory,x, tmpwb, datax,Tick) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(x,indexc,L2CacheState,shared) and
update_msi(y,indexc,L2CacheState,csy) and
update_msi(z,indexc,L2CacheState,csz)
}
else { /* write */
stringx = {if tag = tagx then "Write Hit" else "Write Miss"} and
stringy = {if tag = tagy then "Write Hit" else "Write Miss"} and
stringz = {if tag = tagz then "Write Hit" else "Write Miss"} and
if tag = tagx then { /* write hit cache x */
/*stringy = " " and*/
/*stringz = " " and*/
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(y,indexc,L2CacheState,csy) and
update_msi(z,indexc,L2CacheState,csz) and
update_msi(x,indexc,L2CacheState,modified)
} else if csx = modified then {
/*stringy = " " and*/
/*stringz = " " and*/
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
memory_unchanged(MainMemory) and
update_msi(y,indexc,L2CacheState,csy) and
update_msi(z,indexc,L2CacheState,csz) and
update_msi(x,indexc,L2CacheState,modified)
} else if csx = invalid then {
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
update_msi(x,indexc,L2CacheState,modified) and
if csy = modified and tagy \neq tag then {
/*stringy = "Write miss" and*/
/*stringz = " " and*/
```

248
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
#define L2CacheMemory[y][indexc] and
#define tmpwb = 8*tagy+indexc and
#define format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, y, indexc, tagy, datay, tmpwb) and
#define write_to_memory(MainMemory,y, tmpwb, datay,Tick)
#define assert (csz = modified and tagz \n\n= tag then {
/*stringz = "Write miss" and*/
dataz = L2CacheMemory[z][indexc] and
#define tmpwb = 8*tagz+indexc and
#define format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
#define write_to_memory(MainMemory,z, tmpwb, dataz,Tick)
#define assert (csz = modified and tagz \n\n= tag then {
/*stringz = "Write miss" and*/
dataz = L2CacheMemory[z][indexc] and
#define tmpwb = 8*tagz+indexc and
#define format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
#define write_to_memory(MainMemory,z, tmpwb, dataz,Tick)
#define assert (csz = modified and tagz \n\n= tag then {
/*stringz = "Write miss" and*/
dataz = L2CacheMemory[z][indexc] and
#define tmpwb = 8*tagz+indexc and
#define format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
#define write_to_memory(MainMemory,z, tmpwb, dataz,Tick)
#define assert (csz = modified and tagz \n\n= tag then {
/*stringz = "Write miss" and*/
dataz = L2CacheMemory[z][indexc] and
#define tmpwb = 8*tagz+indexc and
#define format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
#define write_to_memory(MainMemory,z, tmpwb, dataz,Tick)
```
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

datax = L2CacheMemory[x][indexc] and
tmpwb = 8*tagx+indexc and
format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, x, indexc, tagx, datax, tmpwb) and
write_to_memory(MainMemory,x, tmpwb, datax,Tick) and
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,y) and
cache_unchanged(L2CacheMemory,L2CacheTag,Vbit,z) and
update_msi(y,indexc,L2CacheState,csy) and
update_msi(z,indexc,L2CacheState,csz) and
update_msi(x,indexc,L2CacheState,modified)
} else if csx = invalid then {
write_to_cache(L2CacheMemory,L2CacheTag,Vbit,x,indexc,DATA,tag,j) and
update_msi(x,indexc,L2CacheState,modified) and
if csy = modified and tagy \n\neq tag then {
/*stringy = "Write miss" and*/
/*stringz = " " and*/
datay = L2CacheMemory[y][indexc] and
tmpwb = 8*tagy+indexc and
format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, y, indexc, tagy, datay, tmpwb) and
write_to_memory(MainMemory,y, tmpwb, datay,Tick)
}

} else {
/*stringy = " " and*/
if csz = modified and tagz \n\neq tag then {
/*stringz = "Write miss" and*/
dataz = L2CacheMemory[z][indexc] and
tmpwb = 8*tagz+indexc and
format("State %d: Processor %t, write-back cache[%t] with tag %t and
data %t to memory[%t] \n", j, z, indexc, tagz, dataz, tmpwb) and
write_to_memory(MainMemory,z, tmpwb, dataz,Tick)
}

} else {
/*stringz = " " and*/
memory_unchanged(MainMemory)
}

}) and
update_msi(y,indexc,L2CacheState,invalid) and
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
893 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
894 update_msi(z, indexc, L2CacheState, invalid) and
895 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z)
906 else {
907 /*stringy = " " and*/
908 /*stringz = " " and*/
909 write_to_cache(L2CacheMemory, L2CacheTag, Vbit, x, indexc, DATA, tag, j) and
910 update_msi(x, indexc, L2CacheState, modified) and
911 memory_unchanged(MainMemory) and
912 update_msi(y, indexc, L2CacheState, invalid) and
913 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, y) and
914 update_msi(z, indexc, L2CacheState, invalid) and
915 cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, z)
916 }
917 }
918 }
919 and
920 if RW = 0 then {
921 format("State %d: Global is receiving from Processor %d: a read request for Address: %d with
922 Data %d, Global State: %d\n", j, x, ADDR, DATA, Tick)
923 } else {
924 format("State %d: Global is receiving from Processor %d: a write request for Address: %d with
925 Data: %d, Global State: %d\n", j, x, ADDR, DATA, Tick)
926 }
927
928 and cmx = L2CacheMemory[x][indexc]
929 and cmy = L2CacheMemory[y][indexc]
930 and cmz = L2CacheMemory[z][indexc]
931 and ncmx = next(L2CacheMemory[x][indexc])
932 and ncmx = next(L2CacheMemory[y][indexc])
933 and ncmz = next(L2CacheMemory[z][indexc])
934 and mm = MainMemory[indexm]
935 and nmm = next(MainMemory[indexm])
936 and header_out() and
937
938 format(" | %2d(%1d) | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] |
939 Memory[%7s] --->%4t |\n", x, x, ADDR, msb_7bits_addr(ADDR), index_field_cache_8(indexc),
940 vbitx, dbitx, tagx, stringx, cmx, cs_text(csx), x, msb_7bits_addr(indexm), mm) and
941 format(" | %2d(%1d) | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] |
```

251
APPENDIX B. TEMPURA CODE FOR CACHE CONTROLLER

932 Memory[%7s] --->%4t |
933 vbity,dbity,tagy,stringy,cmy,cs_text(csy),y,msb_7bits_addr(indexm),mm) and
934 format("| %2d(%1d) | %d | %4t | %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] |
935 Memory[%7s] --->%4t |
936 vbitz,dbitz,tagz,stringz,cmz,cs_text(csz),z,msb_7bits_addr(indexm),mm) and
937 format("| %2d(%1d) | %d | %4t | %7s | Cache[%3s] | %t | %t | %2t | | %3t | %10s[%1d] |
938 Memory[%7s] --->%4t |
939 nvbitx,ndbitx,ntagx,ncmx,cs_text(ncsx),x,msb_7bits_addr(indexm),nmm) and
940 format("| %2d(%1d) | %d | %4t | %7s | Cache[%3s] | %t | %t | %2t | | %3t | %10s[%1d] |
941 Memory[%7s] --->%4t |
942 nvbity,ndbity,ntagy,ncmy,cs_text(ncsy),y,msb_7bits_addr(indexm),nmm) and
943 format("| %2d(%1d) | %d | %4t | %7s | Cache[%3s] | %t | %t | %2t | | %3t | %10s[%1d] |
944 Memory[%7s] --->%4t |
945 nvbitz,ndbitz,ntagz,ncmz,cs_text(ncsz),z,msb_7bits_addr(indexm),nmm) and
946 format("| %2d(%1d) | %d | %4t | %7s | Cache[%3s] | %t | %t | %2t | | %3t | %10s[%1d] |
947 tcl("HM",[x,y,z,stringx,stringy,stringz,indexc,j,nvbitx,ndbitx,ndbity,ndbitz,
948 DATA,Tick,nmm,ADDR,ADDR mod 16]) and Sx = 
949 "["+str(x)+","+str(RW)+","+str(ADDR)+"]" and /*output(Sx) and*/
955 Sy = 
956 "["+str(y)+","+str(RW)+","+str(ADDR)+"]" and /*output(Sy) and*/
959 Sz = 
965 "["+str(z)+","+str(RW)+","+str(ADDR)+"]" and /*output(Sz) and*/

252
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

",""+stringz+""+","+str(cmz)+","+\""+cs_text(csz)+""+","+\""+cs_text(ncsz)+""+","+\""+msb_7bits_addr(indexm)+""+","+str(mm)+","+str(nmm)+"]" and /*output(Sz) and*/

prog_send2(x,Sx+".") and prog_send2(y,Sy+".") and prog_send2(z,Sz+".") and footer_out(ncsx) and if j<9 then {next(always break)} else skip and

header_property(ncsx) and
Invalid_State(cmx,ncmx,cmy,ncmy,cmz,ncmz,mm,nmm,x,y,z,ncsx,ncsy,ncsz) and
Shared_State(cmx,ncmx,cmy,ncmy,cmz,ncmz,mm,nmm,x,y,z,ncsx,ncsy,ncsz) and
Consistency_Property(ncmx,ncmy,cmz,ncmz,mm,nmm,x,y,z,ncsx,ncsy,ncsz) and
MSI_Protocol(ncsx,ncsy,ncsz,x,y,z) and
Global_State_Consistency(x,y,z,j,Tick,ncsx,ncsy,ncsz) and
Check_Read_Write_Miss_Hit(RW,x,y,z,ADDR,tag,tagx,tagy,tagz,csx,csy,csz,stringx,stringy,stringz,ncsx)
}
}
define Global_State_Consistency(x,y,z,j,Tick,ncsx,ncsy,ncsz) ={
skip and
if ncsx=shared or ncsx=modified or ncsx=invalid then {
if j=Tick then {
/*PID=strint(aval1(T,0)) and
RW = strint(aval1(T,1)) and
addr=strint(aval1(T,2)) and
dato = strint(aval1(T,3)) and
Tick=strint(aval1(T,4)) and */
footer_property_1(ncsx) and

format("| Global State Check | Global = %2d | Pass |\n",j) and
format("| Local State Check | Active[%d] = %2d | Pass |\n",x,Tick) and
format("| Local State Check | Idle[%d] = %2d | Pass |\n",y,Tick) and
format("| Local State Check | Idle[%d] = %2d | Pass |\n",z,Tick)
} else {
footer_property_1(ncsx) and

format("| Global State Check | Global = %2d | Fail |\n",j) and
format("| Local State Check | Active[%d] = %2d | Fail |\n",x,Tick) and
format("| Local State Check | Idle[%d] = %2d | Fail |\n",y,Tick) and
format("| Local State Check | Idle[%d] = %2d | Fail |\n",z,Tick)
}
}
}
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

define Invalid_State(cmx,ncmx,cmy,ncmy,cmz,ncmz,mm,nmm,x,y,z,ncsx,ncsy,ncsz) = {
    skip and
    if ncsx=invalid then {
        if ncmx = -8 or (ncmx = nmm and ncsx ≠ shared and (ncmx ≠ ncmy or ncmx ≠ ncmz)) then {
            format("| Invalid State Check | %10s[%1d] | Pass |\n",cs_text(ncsx),x)
        } else {
            format("|Invalid State Check | %10s[%1d] | Fail |\n",cs_text(ncsx),x)
        }
    } else {
        format("|Invalid State Check | %10s[%1d] | NA |\n",cs_text(ncsx),x)
    } and
    if ncsy=invalid then {
        if ncmy = -8 or (ncmy = nmm and ncsy ≠ shared and (ncmy ≠ ncmx or ncmy ≠ ncmz)) then {
            format("| Invalid State Check | %10s[%1d] | Pass |\n",cs_text(ncsy),y)
        } else {
            format("|Invalid State Check | %10s[%1d] | Fail |\n",cs_text(ncsy),y)
        }
    } else {
        format("|Invalid State Check | %10s[%1d] | NA |\n",cs_text(ncsy),y)
    } and
    if ncsz=invalid then {
        if ncmz = -8 or (ncmz = nmm and ncsz ≠ shared and (ncmz ≠ ncmx or ncmz ≠ ncmx)) then {
            format("| Invalid State Check | %10s[%1d] | Pass |\n",cs_text(ncsz),z)
        } else {
            format("|Invalid State Check | %10s[%1d] | Fail |\n",cs_text(ncsz),z)
        }
    } else {
        format("|Invalid State Check | %10s[%1d] | NA |\n",cs_text(ncsz),z)
    }
}.}

define Shared_State(cmx,ncmx,cmy,ncmy,cmz,ncmz,mm,nmm,x,y,z,ncsx,ncsy,ncsz) = {
    skip and
    if ncsx=shared then {
        if ncmx = nmm or ncmx = ncmy and ncmax = ncmx and ncmx ≠ -8 then {
            format("| Shared State Check | %10s[%1d] | Pass |\n",cs_text(ncsx),x)
        } else {

```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
format("| Shared State Check | %10s[%1d] | Fail |\n", cs_text(ncsx), x)
}
} else {
    format("| Shared State Check | %10s[%1d] | NA |\n", cs_text(ncsx), x)
} and
    if ncsy = shared then {
        if ncmy = nmm or ncmy = ncmx and ncmy = ncmz and ncmy \neq -8 then {
            format("| Shared State Check | %10s[%1d] | Pass |\n", cs_text(ncsy), y)
        } else {
            format("| Shared State Check | %10s[%1d] | Fail |\n", cs_text(ncsy), y)
        }
    } else {
        format("| Shared State Check | %10s[%1d] | NA |\n", cs_text(ncsy), y)
    } and
    if ncsz = shared then {
        if ncmz = nmm or ncmz = ncmx and ncmz = ncmy and ncmz \neq -8 then {
            format("| Shared State Check | %10s[%1d] | Pass |\n", cs_text(ncsz), z)
        } else {
            format("| Shared State Check | %10s[%1d] | Fail |\n", cs_text(ncsz), z)
        }
    } else {
        format("| Shared State Check | %10s[%1d] | NA |\n", cs_text(ncsz), z)
    }
}

define Consistency_Property(ncmx, ncmy, ncmz, nmm, x, y, z, ncsx, ncsy, ncsz) = {
    skip and
    /* If the cache consistent with the main memory */
    if ncsx = shared then {
        if ncmx = nmm then {
            format("| Consistency Property Check | %10s[%1d] | Pass |\n", cs_text(ncsx), x)
        } else {
            format("| Consistency Property Check | %10s[%1d] | Fail |\n", cs_text(ncsx), x)
        }
    } else {
        format("| Consistency Property Check | %10s[%1d] | NA |\n", cs_text(ncsx), x)
    } and
    if ncsy = shared then {
```

255
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```
if ncmy = nmm then {
    format("| Consistency Property Check | %10s[%1d] | Pass |
\n", cs_text(ncsy), y)
} else {
    format("| Consistency Property Check | %10s[%1d] | Fail |
\n", cs_text(ncsy), y)
}
else {
    format("| Consistency Property Check | %10s[%1d] | NA |
\n", cs_text(ncsy), y)
}
and
if ncsz=shared then {
    if ncmz = nmm then {
        format("| Consistency Property Check | %10s[%1d] | Pass |
\n", cs_text(ncsz), z)
    } else {
        format("| Consistency Property Check | %10s[%1d] | Fail |
\n", cs_text(ncsz), z)
    }
} else {
    format("| Consistency Property Check | %10s[%1d] | NA |
\n", cs_text(ncsz), z)
}
}

define header_property(ncsx) = {
    if ncsx = shared or ncsx = modified or ncsx = invalid then {
        format("-------------------------------------------------------------------
") and
        format("| Property | PID | Result |
\n") and
        format("-------------------------------------------------------------------
")
    }
}

define footer_property_1(ncsx) = {
    if ncsx = shared or ncsx = modified or ncsx = invalid then {
        format("-------------------------------------------------------------------
")
    }
}

define footer_property() = {
    format("-------------------------------------------------------------------
")
}

define MSI_Protocol(ncsx, ncsy, ncsz, x, y, z) = {
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
skip and
/* Allowed MSI States */
if (ncsx = invalid and ncsy = invalid and ncsz = invalid) or
  (ncsx = invalid and ncsy = invalid and ncsz = modified) or
  (ncsx = invalid and ncsy = invalid and ncsz = shared) or
  (ncsx = invalid and ncsy = modified and ncsz = invalid) or
  (ncsx = invalid and ncsy = shared and ncsz = invalid) or
  (ncsx = invalid and ncsy = shared and ncsz = shared) or
  (ncsx = modified and ncsy = invalid and ncsz = invalid) or
  (ncsx = shared and ncsy = invalid and ncsz = invalid) or
  (ncsx = shared and ncsy = invalid and ncsz = shared) or
  (ncsx = shared and ncsy = shared and ncsz = invalid) or
  (ncsx = shared and ncsy = shared and ncsz = shared) then {
  footer_property() and
  format("| | %10s[%1d] | |
",cs_text(ncsx),x) and
  format("| MSI Protocol Check | %10s[%1d] | Pass |
",cs_text(ncsy),y) and
  format("| | %10s[%1d] | |
",cs_text(ncsz),z) /*and
  footer_property()*/
} else {
  footer_property() and
  format("| | %10s[%1d] | |
",cs_text(ncsx),x) and
  format("| MSI Protocol Check | %10s[%1d] | Fail |
",cs_text(ncsy),y) and
  format("| | %10s[%1d] | |
",cs_text(ncsz),z) /*and
  footer_property()*/
}
}
```

```c
define Check_Read_Write_Miss_Hit(RW,x,y,z,ADDR,tag,tagx,tagy,tagz,
  csx,csy,csz,stringx,stringy,stringz,ncsx) =
  { exists Estring00,Estring01,Estring10,Estring11 : {
    skip and
    Estring00="Read Miss" and Estring01="Read Hit" and
    Estring10="Write Miss" and Estring11="Write Hit" and
    if ncsx=shared or ncsx=modified or ncsx=invalid then {
      if RW=0 then { /* RW=0 Read Check */
        if tag ≠ tagx and (csx ≠ shared or csx ≠ modified) and
          tag ≠ tagy and (csy ≠ shared or csy ≠ modified) and
          tag ≠ tagz and (csz ≠ shared or csz ≠ modified) then {
            if stringx=Estring00 and stringy=Estring00 and stringz=Estring00 then {
```
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

footer_property() and
format("| | %12t[%d] | \n",stringx,x) and
format("| Read Miss Check | %12t[%d] | Pass |\n",stringy,y) and
format("| | %12t[%d] | \n",strings,z) and
footer_property()
}
else {

format("| | %12t[%d] | \n",stringx,x) and
format("| Read Miss Check | %12t[%d] | Fail |\n",stringy,y) and
format("| | %12t[%d] | \n",strings,z) and
footer_property()
}
}
else {
if stringx=Estring01 or stringy=Estring01 or stringz=Estring01 then {
footer_property() and
format("| | %12t[%d] | \n",stringx,x) and
format("| Read Hit Check | %12t[%d] | Pass |\n",stringy,y) and
format("| | %12t[%d] | \n",strings,z) and
footer_property()
}
else {
footer_property() and
format("| | %12t[%d] | \n",stringx,x) and
format("| Read Hit Check | %12t[%d] | Fail |\n",stringy,y) and
format("| | %12t[%d] | \n",strings,z) and
footer_property()
}
}
else { /* RW=1 Write Check */
if tag \neq tagx and tag \neq tagy and tag \neq tagz then {
if stringx=Estring10 and stringy=Estring10 and stringz=Estring10 then {
footer_property() and
format("| | %12t[%d] | \n",stringx,x) and
format("| Write Miss Check | %12t[%d] | Pass |\n",stringy,y) and
format("| | %12t[%d] | \n",strings,z) and
footer_property()
}
else {
footer_property() and
format("| | %12t[%d] | \n",stringx,x) and
format("| Write Miss Check | %12t[%d] | Fail |\n",stringy,y) and

APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

format(" | %12t[%d] | \n",strings,z) and footer_property()
} else {
  if stringx=Estring11 or stringy=Estring11 or stringz=Estring11 then {
    footer_property() and
    format(" | %12t[%d] | \n",stringx,x) and
    format(" | Write Hit Check | %12t[%d] | Pass \n",stringy,y) and
    format(" | %12t[%d] | \n",strings,z) and
    footer_property()
  } else {
    footer_property() and
    format(" | %12t[%d] | \n",stringx,x) and
    format(" | Write Hit Check | %12t[%d] | Fail \n",stringy,y) and
    format(" | %12t[%d] | \n",strings,z) and
    footer_property()
  }
}

define write_to_cache(L2CacheMemory,L2CacheTag,Vbit,X,M,V,tag,j) = {
  skip and
  format("State %d: Processor %t writing to Cache[%t] value %t and tag %t\n",j,X,M,V,tag) and
  (forall i<nprocessors :
    (forall j<ncachelocations:
      if i=X and j=M then {
        if Vbit[i][j] = 1 then {stable(Vbit[i][j])}
        else {Vbit[i][j] := 1} and
        L2CacheTag[i][j] := tag and
        L2CacheMemory[i][j]:=V
      } else {
        stable(Vbit[i][j]) and
        stable(L2CacheTag[i][j]) and
        stable(L2CacheMemory[i][j])
      }
    }
  )
}
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
define write_to_memory(MainMemory, X, M, V, Tick) = {
    skip and tcl("MM", [V, msb(M), M, M mod 8, Tick]) and
    format("State %d: Processor %t writing to global memory[%t] value %t\n", Tick, X, M, V) and
    (forall j<nmemorylocations: {
        if j=M then {MainMemory[j]:=V}
        else { stable(MainMemory[j]) }
    })
}

define memory_unchanged(MainMemory) = {
    skip and
    (forall j<nmemorylocations: {
        stable(MainMemory[j])
    })
}

define cache_unchanged(L2CacheMemory, L2CacheTag, Vbit, x) = {
    skip and
    (forall j<ncachelocations: {
        stable(Vbit[x][j]) and
        stable(L2CacheTag[x][j]) and
        stable(L2CacheMemory[x][j])
    })
}

define header_out() = {
    format("-----------------------------------------------------------------------------------------------
    | Pid | Operation | Addr | Binary Addr | Cache[Index] | Valid Bit | Dirty Bit |...
    | Tag | Hit-Miss | Data | Coherence State | Memory[..Addr..] ----> Data |\n    ")
}
```

260
define footer_out(ncsx) = {
    if ncsx=shared or ncsx=modified or ncsx=invalid then {
        format("--------------------------------------------------------------------------------------------------
        ------------------------------------------------------------------------------\n")
    }
}

define get_var3(MainMemory,L2CacheMemory,L2CacheTag,L2CacheState,Vbit,Dbit,j) = {
    exists T,PID,addr,tag,index8,B,v,z,Tick,RW,datato,datato : {
        get2(T) and
        PID=strint(aval1(T,0)) and
        RW = strint(aval1(T,1)) and
        addr=strint(aval1(T,2)) and
        datato = strint(aval1(T,3)) and
        Tick=strint(aval1(T,4)) and
        format("\n\n") and
        if RW = 0 then {
            cpu_request(MainMemory,L2CacheMemory,L2CacheTag,L2CacheState,Vbit,Dbit,PID,RW,addr,datato,j,Tick)
        } else {
            cpu_request(MainMemory,L2CacheMemory,L2CacheTag,L2CacheState,Vbit,Dbit,PID,RW,addr,datato,j,Tick)
        }
        and
        tcl("tmr",[Tick]) and
        tcl("CPUREQ",[PID,Tick,msb(addr),RW,datato,j]) and
        forall i<32 :(tcl("ABCD",[i,msb32_2(i,addr)]))
    }
}

if PID=0 then {
    tcl("CM",[0,Tick,datato,tag_field_cache(addr),index_field_cache_8(addr mod 8),
    addr mod 8,msb_14_index(addr mod 16),addr mod 16,RW])
} else if PID=1 then {
    tcl("CM",[1,Tick,datato,tag_field_cache(addr),index_field_cache_8(addr mod 8),
    addr mod 8,msb_14_index(addr mod 16),addr mod 16,RW])
} else if PID=2 then {

APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
tcl("CM",[2,Tick,datato,tag_field_cache(addr),index_field_cache_8(addr mod 8),
addr mod 8,msb_14_index(addr mod 16),addr mod 16,RW])
}

set print_states=false.

/* run */ define L2_Cache_P0_P1_v0() = {
exists PID,B,addr,tag,index8,j,i,MainMemory, CacheMemory, L2CacheTag,
L2CacheMemory, Core, Tag, Index, Timer, L2CacheState, Vbit, Dbit,Select :
list(MainMemory, nmemorylocations) and stable(struct(MainMemory)) and
list(CacheMemory, ncachelocations) and stable(struct(CacheMemory)) and
list(Core, nprocessors) and stable(struct(Core)) and
list(L2CacheMemory,nprocessors) and stable(struct(L2CacheMemory)) and
list(L2CacheTag,nprocessors) and stable(struct(L2CacheTag)) and
list(L2CacheState,nprocessors) and stable(struct(L2CacheState)) and
list(Vbit,nprocessors) and stable(struct(Vbit)) and
list(Dbit,nprocessors) and stable(struct(Dbit)) and
(forall i<nprocessors:
list(L2CacheMemory[i], ncachelocations) and stable(struct(L2CacheMemory[i])) and
list(L2CacheTag[i], ncachelocations) and stable(struct(L2CacheTag[i])) and
list(L2CacheState[i], ncachelocations) and stable(struct(L2CacheState[i])) and
list(Vbit[i], ncachelocations) and stable(struct(Vbit[i])) and
list(Dbit[i], ncachelocations) and stable(struct(Dbit[i]))
)
and
list(Tag, nlocations) and stable(struct(Tag)) and
list(Index, nlocations) and stable(struct(Index)) and

{prog_send1(0,"load 'Processor_0_5'.") and
prog_send1(1,"load 'Processor_1_5'.") and
prog_send1(2,"load 'Processor_2_5'.");skip;

{prog_send1(0,"run L2_Processor_0().") and
prog_send1(1,"run L2_Processor_1().") and
prog_send1(2,"run L2_Processor_2().");skip;
```

262
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```
{ tcl("init", [2,8]) and always tclbreak() and
(forrall j<nmemorylocations : MainMemory[j] = initial_value2) and
(forrall j<ncachelocations : CacheMemory[j] = initial_value) and
(forrall j<nprocessors : Core[j] = initial_value) and
(forrall j<ncachelocations : (L2CacheMemory[i][j] = initial_value and
L2CacheTag[i][j] = -1 and
L2CacheState[i][j] = invalid and
Vbit[i][j] = 0 and
Dbit[i][j] = 0)
)
)
)
(forrall j<nlocations : Tag[j] = initial_value) and
(forrall j<nlocations : Index[j] = initial_value) and
Timer = 0 and
(forrall j<ncachelocations : tcl("INDEX", [j, index_field_cache_8(j)])) and
for j<10 do {
(forrall i<nmemorylocations : tcl("IM", [MainMemory[i], msb(i), i mod 16])) and
{Select = Random mod 3 and
prog_send_ne1(0,"["+str(Select)+",","+str(j)+"]+".) and
prog_send_ne1(1,"["+str(Select)+",","+str(j)+"]+".) and
prog_send_ne1(2,"["+str(Select)+",","+str(j)+"]+".)
};
{skip and get var3(MainMemory, L2CacheMemory, L2CacheTag, L2CacheState, Vbit, Dbit, j)}
}
}
{prog_send1(0,"exit.") and
prog_send1(1,"exit.") and
prog_send1(2,"exit.")
}
}
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

Listing B.2: Tempura Code for Processor 0

```c
/* -*- Mode: C -*-
 *
* Processor_0_5.t
 *
* This file is part Tempura: Interval Temporal Logic interpreter.
 *
* Copyright (C) 1998-2017 Nayef Alshammari, Antonio Cau
 *
* Tempura is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
 *
* Tempura is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
 *
* You should have received a copy of the GNU General Public License
* along with Tempura. If not, see <http://www.gnu.org/licenses/>.
 *
*/

load "./library/conversion".
load "./library/tcl".
load "./library/exprog".

define avar1(X,a) =
{
    X[a]
}.

define avall(X,b) =
{
    X[b]
}.
```

264
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

38 define atime1(X,c) =
39 {
40   strint(X[c])
41 }.
42
define atime_micro1(X,d) =
43 {
44   strint(X[d])
45 }.
46
define get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
47 NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,State) = {
48   header_out() and
49   format("| %2d | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] | ...
50      Memory[%7s] ---> %4t ...
51    
52   format("| %2d | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] | ...
53      Memory[%7s] ---> %4t ...
54    
55   footer_out() and
56   header_property() and
57   Consistency_Property(PID,NCMX,NCSX,NMM) and
58   Invalid_State(PID,NCMX,NCSX,NMM) and
59   Shared_State(PID,NCMX,NCSX,NMM) and
60   /* Local_State_Consistency(PID,State) and */
61   Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) and
62   footer_property()
63 }.
64
define Local_State_Consistency(PID,State) ={
65   if State = 0 then {
66     format("| Local State Check | Active[%d] = %2d | Pass |\n",PID,State)
67   } else{
68     if prev(State) = State-1 then {
69
define Consistency_Property(PID, NCMX, NCSX, NMM) = {
/* If the cache consistent with the main memory */
if NCSX="Shared" then {
  if NCMX = NMM then {
    format("| Consistency Property Check | %10s[%1d] | Pass |\n", NCSX, PID)
  } else {
    format("| Consistency Property Check | %10s[%1d] | Fail |\n", NCSX, PID)
  }
} else {
  format("| Consistency Property Check | %10s[%1d] | NA |\n", NCSX, PID)
}
}.

define Invalid_State(PID, NCMX, NCSX, NMM) = {
  if NCSX="Invalid" then {
    if NCMX = -8 or
    NCMX = NMM and NCSX ≠ "Shared" then {
      format("| Invalid State Check | %10s[%1d] | Pass |\n", NCSX, PID)
    } else {
      format("| Invalid State Check | %10s[%1d] | Fail |\n", NCSX, PID)
    }
  } else {
    format("| Invalid State Check | %10s[%1d] | NA |\n", NCSX, PID)
  }
}.

define Shared_State(PID, NCMX, NCSX, NMM) = {
  if NCSX="Shared" then {
    if NCMX = NMM or NCMX ≠ -8 then {

format("| Shared State Check | %10s[%1d] | Pass |\n",NCSX,PID)
} else {format("| Shared State Check | %10s[%1d] | Fail |\n",NCSX,PID)}
} else {
format("| Shared State Check | %10s[%1d] | NA |\n",NCSX,PID)
}

define Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) = {
exists Estring00,Estring01,Estring10,Estring11 : {
Estring00="Read Miss" and Estring01="Read Hit" and
Estring10="Write Miss" and Estring11="Write Hit" and

if RW=0 then { /* RW=0 Read Check */
if TAG ≠ TAGX and (CSX ≠ "Shared" or CSX ≠ "Modified") then {
format("| Read Miss Check | %10t[%d] | Pass |\n",STRINGX,PID)
} else {
format("| Read Miss Check | %10t[%d] | Fail |\n",STRINGX,PID)
}
}
else {
if STRINGX=Estring01 then {
format("| Read Hit Check | %10t[%d] | Pass |\n",STRINGX,PID)
} else {
format("| Read Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
}
}
else {
if STRINGX=Estring10 then {
format("| Write Miss Check | %10t[%d] | Pass |\n",STRINGX,PID)
} else {
format("| Write Miss Check | %10t[%d] | Fail |\n",STRINGX,PID)
}
}
}
else {
if STRINGX=Estring11 then {


APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```
format("| Write Hit Check | %10t[%d] | Pass |\n",STRINGX,PID)
} else {
format("| Write Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
}

define header_out() = {
format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
") and
format("| Pid | Operation | Addr | Binary Addr | Cache[Index] | Valid Bit | Dirty Bit |
Tag | Hit-Miss | Data | Coherence State | Memory[..Addr..] ---> Data |\n") and
format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
")
}.

define footer_out() = {
format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
")
}.

define header_property() = {
format("-------------------------------------------------------------------
") and
format("| Property | PID | Result |
") and
format("-------------------------------------------------------------------
")
}.

define footer_property() = {
format("-------------------------------------------------------------------
")
}.

define assert(Pid,j,RW,addr,data,Tick) = {
exists Operation : {
if RW=0 then {Operation="Read" and
format("State %d: Processor %d is sending %6s request from Address: %d, and Data: %d, and ...

```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

Global State: %d

} else {Operation="Write" and

format("State %d: Processor %d is sending %6s request to Address: %d, and Data: %d, and ...

Global State: %d

} and

format("!PROG: assert %d:%d:%d:%d:%d:

and

set print_states=false.

/* run */ define L2_Processor_0() = {

exists j,Tick,PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,Tock,State :

for j<10 do {

{input Tick and skip and State=Tick[1] and

if Tick[0]=0 then {

assert(Tick[0],j,Random mod 2,Random mod 16,Random mod 30, Tick[1])

} else

format("State %d: Processor 0 is idle",Tick[1])

};

{ empty and

input Tock and output(Tock) and

PID=Tock[0] and RW=Tock[1] and DADDR=Tock[2] and BADDR=Tock[3] and INDEXC=Tock[4] and
VBITX=Tock[5] and
NTAGX=Tock[11] and
STRINGX=Tock[12] and CMX=Tock[13] and NCMX=Tock[14] and CSX=Tock[15] and NCSX=Tock[16] and
INDEXM=Tock[17] and
NMM=Tock[18] and NMMM=Tock[19] and

get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,State)

}

}

}.

269
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

Listing B.3: Tempura Code for Processor 1

```c
/* -*- Mode: C -*-
 *
* Processor_1_5.t
 *
* This file is part Tempura: Interval Temporal Logic interpreter.
 *
* Copyright (C) 1998-2017 Nayef Alshammari, Antonio Cau
 *
* Tempura is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
 *
* Tempura is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
 *
* You should have received a copy of the GNU General Public License
* along with Tempura. If not, see <http://www.gnu.org/licenses/>.
 *
*/

load "../library/conversion".
load "../library/tcl".
load "../library/exprog".

define avar1(X,a) =
  {
    X[a]
  }.

define avall(X,b) =
  {
    X[b]
  }.
```

270
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
38 define atime1(X,c) =
39 { strint(X[c]) }. define atime_micro1(X,d) =
40 { strint(X[d]) }. define get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
41 NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,NMM,State) = {
42 header_out() and format("| %2d | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] |
... Memory[%7s] ---> %4t ...
\n",PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAGX,STRINGX,CMX,NCMX,CSX,PID,INDEXM,NMM) and
43 format("| %2d | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] |
... Memory[%7s] ---> %4t ...
\n",PID,RW,DADDR,BADDR,INDEXC,NVBITX,NDBITX,NTAGX,STRINGX,CMX,NCMX,PID,INDEXM,NMM) and
44 footer_out() and header_property() and Consistency_Property(PID,NCMX,NCSX,NMM) and
45 Invalid_State(PID,NCMX,NCSX,NMM) and
46 Shared_State(PID,NCMX,NCSX,NMM) and
47 /*Local_State_Consistency(PID,State) and*/
48 Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) and
49 footer_property() }. define Local_State_Consistency(PID,State) ={
50 if State = 0 then {
51 format("| Local State Check | Active[%d] = %2d | Pass \n",PID,State)
52 } else{
53 if prev(State) = State-1 then {

```

APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
    format("| Local State Check | Active[%d] = %2d | Pass |\n",PID,State)
  } else {
    format("| Local State Check | Active[%d] = %2d | Fail |\n",PID,State)
  }
}

define Consistency_Property(PID,NCMX,NCSX,NMM) = {
  /* If the cache consistent with the main memory */
  if NCSX="Shared" then {
    if NCMX = NMM then {
      format("| Consistency Property Check | %10s[%1d] | Pass |\n",NCSX,PID)
    } else {
      format("| Consistency Property Check | %10s[%1d] | Fail |\n",NCSX,PID)
    }
  } else {
    format("| Consistency Property Check | %10s[%1d] | NA |\n",NCSX,PID)
  }
}

define Invalid_State(PID,NCMX,NCSX,NMM) = {
  if NCSX="Invalid" then {
    if NCMX = -8 or
    NCMX = NMM and NCSX ≠ "Shared" then {
      format("| Invalid State Check | %10s[%1d] | Pass |\n",NCSX,PID)
    } else {
      format("| Invalid State Check | %10s[%1d] | Fail |\n",NCSX,PID)
    }
  } else {
    format("| Invalid State Check | %10s[%1d] | NA |\n",NCSX,PID)
  }
}

define Shared_State(PID,NCMX,NCSX,NMM) = {
  if NCSX="Shared" then {
    if NCMX = NMM or NCMX ≠ -8 then {
      format("| Shared State Check | %10s[%1d] | Pass |\n",NCSX,PID)
    } else {
      format("| Shared State Check | %10s[%1d] | Fail |\n",NCSX,PID)
    }
  }
```

272
APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
} else {
    format("| Shared State Check | %10s[%1d] | NA |\n",NCSX,PID);
}
}

define Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) = {
    exists Estring00,Estring01,Estring10,Estring11 : {
        Estring00="Read Miss" and Estring01="Read Hit" and
        Estring10="Write Miss" and Estring11="Write Hit" and

        if RW=0 then { /* RW=0 Read Check */
            if TAG != TAGX and (CSX != "Shared" or CSX != "Modified") then {
                if STRINGX=Estring00 then {
                    format("| Read Miss Check | %10t[%d] | Pass |\n",STRINGX,PID)
                } else {
                    format("| Read Miss Check | %10t[%d] | Fail |\n",STRINGX,PID)
                }
            } else {
                if STRINGX=Estring01 then {
                    format("| Read Hit Check | %10t[%d] | Pass |\n",STRINGX,PID)
                } else {
                    format("| Read Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
                }
            }
        } else { /* RW=1 Write Check */
            if TAG != TAGX then {
                if STRINGX=Estring10 then {
                    format("| Write Miss Check | %10t[%d] | Pass |\n",STRINGX,PID)
                } else {
                    format("| Write Miss Check | %10t[%d] | Fail |\n",STRINGX,PID)
                }
            } else {
                if STRINGX=Estring11 then {
                    format("| Write Hit Check | %10t[%d] | Pass |\n",STRINGX,PID)
                } else {
                    format("| Write Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
                }
            }
        }
    }
}
```

APPENDIX B: APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
format("| Write Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
}.
define header_out() = {
    format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
") and
    format("| Pid | Operation | Addr | Binary Addr | Cache[Index] | Valid Bit | Dirty Bit |
    Tag | Hit-Miss | Data | Coherence State | Memory[..Addr...] ---> Data |\n") and
    format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
")
}.
define footer_out() = {
    format("-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
")
}.
define header_property() = {
    format("-------------------------------------------------------------------
") and
    format("| Property | PID | Result |
    -------------------------------------------------------------------
")
}.
define footer_property() = {
    format("-------------------------------------------------------------------
")
}.
define assert(Pid,j,RW,addr,data,Tick) = {
    exists Operation : {
        if RW=0 then {Operation="Read" and
            format("State %d: Processor %d is sending %6s request from Address: %d, and Data: %d, and ...
            Global State: %d\n",Tick,Pid,Operation,addr,data,Tick)
        } else {Operation="Write" and
            ...
        }
    }
}.
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```plaintext
format("State %d: Processor %d is sending %6s request to Address: %d, and Data: %d, and ...
Global State: %d\n",Tick,Pid,Operation,addr,data,Tick)
} and
format("!PROG: assert %d:%d:%d:%d:%d:!
",Pid,RW,addr,data,Tick)
}
}

set print_states=false.

/* run */ define L2_Processor_1() = {
exists j,Tick,PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,Tock,State : {
for j<10 do {
{input Tick and skip and State=Tick[1] and
if Tick[0]=1 then {
assert(Tick[0],j,Random mod 2,Random mod 16,Random mod 30, Tick[1])
} else
format("State %d: Processor 1 is idle\n",Tick[1])
};
{ empty and
input Tock and output(Tock) and
PID=Tock[0] and RW=Tock[1] and DADDR=Tock[2] and BADDR=Tock[3] and INDEXC=Tock[4] and
VBITX=Tock[5] and
NTAGX=Tock[11] and
STRINGX=Tock[12] and CMX=Tock[13] and NCMX=Tock[14] and CSX=Tock[15] and NCSX=Tock[16] and
INDEXM=Tock[17] and
MM=Tock[18] and NMM=Tock[19] and
get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,State)
}
}
}
}
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

Listing B.4: Tempura Code for Processor 2

```c
/* -*- Mode: C -*-
 * Processor_2_5.t
 *
 * This file is part Tempura: Interval Temporal Logic interpreter.
 *
 * Copyright (C) 1998-2017 Nayef Alshammari, Antonio Cau
 *
 * Tempura is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Tempura is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Tempura. If not, see <http://www.gnu.org/licenses/>.
 *
 */

load "../library/conversion".
load "../library/tcl".
load "../library/exprog".

define avar1(X,a) =
{
    X[a]
}.

define avall(X,b) =
{
    X[b]
}.
```
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c

38 define atime1(X,c) =
39 {
40     strint(X[c])
41 }.
42
define atime_micro1(X,d) =
43 {
44     strint(X[d])
45 }.
46
define get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
47 NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,NMM,State) = {
48     header_out() and
49     format("| %2d | %d | %4t| %7s | Cache[%3s] | %t | %t | %2t | %10s | %3t | %10s[%1d] | ...
50     Memory[%7s] ---> %4t ... |
51     ",PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAGX,STRINGX,CMX,CSX,PID,INDEXM,NMM) and
52     footer_out() and
53     header_property() and
54     Consistency_Property(PID,NCMX,NCSX,NMM) and
55     Invalid_State(PID,NCMX,NCSX,NMM) and
56     Shared_State(PID,NCMX,NCSX,NMM) and
57     /*Local_State_Consistency(PID,State) and*/
58     Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) and
59     footer_property()
60 }
61
define Local_State_Consistency(PID,State) =
62 if State = 0 then {
63     format("| Local State Check | Active[%d] = %2d | Pass |\n",PID,State)
64 } else {
65     if prev(State) = State-1 then {
66         format("| Local State Check | Active[%d] = %2d | Fail |\n",PID,State)
67     }
68 }
```

277
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
format("| Local State Check | Active[%d] = %2d | Pass |\n",PID,State)
}
}
format("| Local State Check | Active[%d] = %2d | Fail |\n",PID,State)
}
}
define Consistency_Property(PID,NCMX,NCSX,NMM) = {
    /* If the cache consistent with the main memory */
    if NCSX="Shared" then {
        if NCMX = NMM then {
            format("| Consistency Property Check | %10s[%1d] | Pass |\n",NCSX,PID)
        } else {
            format("| Consistency Property Check | %10s[%1d] | Fail |\n",NCSX,PID)
        }
    } else {
        format("| Consistency Property Check | %10s[%1d] | NA |\n",NCSX,PID)
    }
}
}
define Invalid_State(PID,NCMX,NCSX,NMM) = {
    if NCSX="Invalid" then {
        if NCMX = -8 or
            NCMX = NMM and NCSX ≠ "Shared" then {
                format("| Invalid State Check | %10s[%1d] | Pass |\n",NCSX,PID)
            } else {
                format("| Invalid State Check | %10s[%1d] | Fail |\n",NCSX,PID)
            }
    } else {
        format("| Invalid State Check | %10s[%1d] | NA |\n",NCSX,PID)
    }
}
}
define Shared_State(PID,NCMX,NCSX,NMM) = {
    if NCSX="Shared" then {
        if NCMX = NMM or NCMX ≠ -8 then {
            format("| Shared State Check | %10s[%1d] | Pass |\n",NCSX,PID)
        } else {
            format("| Shared State Check | %10s[%1d] | Fail |\n",NCSX,PID)
        }
    } else {
        format("| Shared State Check | %10s[%1d] | NA |\n",NCSX,PID)
    }
}
```

278
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
} else {
    format("| Shared State Check | %10s[%1d] | Fail |\n", NCSX, PID)
} else {

#define Check_Read_Write_Miss_Hit(RW,PID,TAG,TAGX,CSX,STRINGX) {
exists Estring00,Estring01,Estring10,Estring11 :
Estring00="Read Miss" and Estring01="Read Hit" and
Estring10="Write Miss" and Estring11="Write Hit" and

if RW=0 then { /* RW=0 Read Check */
    if TAG \neq TAGX and (CSX \neq "Shared" or CSX \neq "Modified") then {
        if STRINGX=Estring00 then {
            format("| Read Miss Check | %10t[%d] | Pass |\n", STRINGX, PID)
        } else {
            format("| Read Miss Check | %10t[%d] | Fail |\n", STRINGX, PID)
        }
    } else {
        if STRINGX=Estring01 then {
            format("| Read Hit Check | %10t[%d] | Pass |\n", STRINGX, PID)
        } else {
            format("| Read Hit Check | %10t[%d] | Fail |\n", STRINGX, PID)
        }
    }
} else { /* RW=1 Write Check */
    if TAG \neq TAGX then {
        if STRINGX=Estring10 then {
            format("| Write Miss Check | %10t[%d] | Pass |\n", STRINGX, PID)
        } else {
            format("| Write Miss Check | %10t[%d] | Fail |\n", STRINGX, PID)
        }
    } else {
        if STRINGX=Estring11 then {
            format("| Write Hit Check | %10t[%d] | Pass |\n", STRINGX, PID)
        } else {
            if STRINGX=Estring10 then {
                format("| Write Hit Check | %10t[%d] | Pass |\n", STRINGX, PID)
            }
        }
    }
}
```

279
APPENDIX B. APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

```c
} else {
    format("| Write Hit Check | %10t[%d] | Fail |\n",STRINGX,PID)
}

define header_out() = {
    format("-------------------------------------------------------------------------
    -------------------------------------------------------------------------------------------------------\n")
    and
    format("| Pid | Operation | Addr | Binary Addr | Cache[Index] | Valid Bit | Dirty Bit |
    Tag | Hit-Miss | Data | Coherence State | Memory[..Addr..] ---> Data |\n")
    and
    format("-------------------------------------------------------------------------
    -------------------------------------------------------------------------------------------------------\n")
}

define footer_out() = {
    format("-------------------------------------------------------------------------
    -------------------------------------------------------------------------------------------------------\n")
}

define header_property() = {
    format("-------------------------------------------------------------------\n")
    and
    format("| Property | PID | Result |
    -------------------------------------------------------------------\n")
}

define footer_property() = {
    format("-------------------------------------------------------------------\n")
}

define assert(Pid,j,RW,addr,data,Tick) = {
    exists Operation : {
        if RW=0 then {
            Operation="Read" and
            format("State %d: Processor %d is sending %6s request from Address: %d, and Data: %d, and ...
                  Global State: %d\n",Tick,Pid,Operation,addr,data,Tick)
        }
    }
```
APPENDIX B: APPENDIX B: TEMPURA CODE FOR CACHE CONTROLLER

188 } else {Operation="Write" and
189 format("State %d: Processor %d is sending %6s request to Address: %d, and Data: %d, and ... Global State: %d\n",Tick,Pid,Operation,addr,data,Tick)
190 } and
191 format("!PROG: assert %d:%d:%d:%d:%d:!
",Pid,RW,addr,data,Tick)
192 }.
193
194 set print_states=false.
195 /* run */ define L2_Processor_2() = {
196 exists j,Tick,PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,TAGX,NVBITX,
197 NDBITX,NTAGX,STRINGX,CMX,NCMX,CSX,NCSX,INDEXM,MM,NMM,Tock,State : { 
198 for j<10 do {
199  {input Tick and skip and State=Tick[1] and
200   if Tick[0]=2 then {
201     assert(Tick[0],j,Random mod 2,Random mod 16,Random mod 30, Tick[1])
202   } else
203     format("State %d: Processor 2 is idle\n",Tick[1])
204  };
205  { empty and
206    input Tock and output(Tock) and
207    PID=Tock[0] and RW=Tock[1] and DADDR=Tock[2] and BADDR=Tock[3] and INDEXC=Tock[4] and
208    VBITX=Tock[5] and
210    NTAGX=Tock[11] and
211    STRINGX=Tock[12] and CMX=Tock[13] and NCMX=Tock[14] and CSX=Tock[15] and NCSX=Tock[16] and
212    INDEXM=Tock[17] and
213    MM=Tock[18] and NMM=Tock[19] and
214    get_var(PID,RW,DADDR,BADDR,INDEXC,VBITX,DBITX,TAG,CMX,INDEXM,MM,NMM,State)
215  })
216  }
217  }
218  }
219  }
220  }
221  }
222  }
223  }
224  }.
281
Appendix C

Appendix C: Tcl/tk Code for Cache Controller
# L2_Cache_MSI_v3.tcl --
#
# Copyright (C) 1998-2017 Nayef H. Alshammari, Antonio Cau
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
#
package provide L2CacheControllerMSI 1.0

namespace eval ::out {
    namespace export -clear *
    namespace export -clear *
    variable canv;

    proc init {n} {
        variable canv;

        wm geometry .top23 1700x800+1+20
        $canv delete all
        $canv config -scrollregion "10 10 1700 1000"
        $canv configure -background black
        #CPU 0
        $canv create rect \
APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
10 10 \n425 455 \n-film black -outline black -width 2 -tags rect0

#State
$canv create text 30 20\n-anchore c -fill red -text State

$canv create text 72 20\n-anchore c -fill red -text Oper.

$canv create text 240 20\n-anchore c -fill red -text Address

$canv create text 405 20\n-anchore c -fill red -text Data

#State
for {set i 0} {$i<10} {incr i} {
$canv create rect \n10 [expr 30+15*$i]\n50 [expr 45+15*$i]\n-width 1 -fill black -outline #00FF00
$canv create text 30 [expr 37+15*$i]\n-anchore c -fill #00FF00 -text "" -tags StateP0($i)
}

#READ
for {set i 0} {$i<10} {incr i} {
$canv create rect \n50 [expr 30+15*$i]\n95 [expr 45+15*$i]\n-width 1 -fill black -outline #00FF00
$canv create text 72 [expr 37+15*$i]\n-anchore c -fill #00FF00 -text "" -tags operationP0($i)
}

#ADDRESS
for {set i 0} {$i<10} {incr i} {
$canv create rect \n
284
```
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

79 95 \[expr 30+15*$i]\n
80 385 \[expr 45+15*$i]\n
81 -width 1 -fill black -outline #00FF00

82 $canv create text 240 \[expr 37+15*$i]\n
83 -anchor c -text "" -tags cpuaddrtextP0($i)

84 }

85

86 #Data

87 for {set i 0} {$i<10} {incr i} {

88 $canv create rect \n
89 385 \[expr 30+15*$i]\n
90 425 \[expr 45+15*$i]\n
91 -width 1 -fill black -outline #00FF00

92 $canv create text 405 \[expr 37+15*$i]\n
93 -anchor c -fill #00FF00 -text "" -tags DataCPU0($i)

94 }

95

96 #CPU 1

97 $canv create rect \n
98 430 10 \n
99 845 455 \n
100 -fill black -outline black -width 2 -tags rect1

101

102 #State

103 $canv create text 452 20\n
104 -anchor c -fill red -text State

105

106 $canv create text 495 20\n
107 -anchor c -fill red -text Oper.

108

109 $canv create text 655 20\n
110 -anchor c -fill red -text Address

111

112 $canv create text 820 20\n
113 -anchor c -fill red -text Data

114

115 #State

116 for {set i 0} {$i<10} {incr i} {

117 $canv create rect \n

285
APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
# READ
for {set i 0} {$i<10} {incr i} {
    $canv create rect 
    470 [expr 30+15*$i]
    805 [expr 30+15*$i]
    -width 1 -fill black -outline #00FF00
    $canv create text 825 [expr 37+15*$i] -anchor c -fill #00FF00 -text "" -tags DataCPU1($i)
}
```

```tcl
# ADDRESS
for {set i 0} {$i<10} {incr i} {
    $canv create rect 
    515 [expr 30+15*$i]
    805 [expr 30+15*$i]
    -width 1 -fill black -outline #00FF00
    $canv create text 660 [expr 37+15*$i] -anchor c -text "" -tags cpuaddrtextP1($i)
}
```

```tcl
# Data
for {set i 0} {$i<10} {incr i} {
    $canv create rect 
    805 [expr 30+15*$i]
    845 [expr 30+15*$i]
    -width 1 -fill black -outline #00FF00
    $canv create text 825 [expr 37+15*$i] -anchor c -fill #00FF00 -text "" -tags DataCPU1($i)
}
```

```tcl
# CPU 2
$canv create rect 
850 10 
```

APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

1270 455 
-fillet black -outline black -width 2 -tags rect2

#State
$canv create text 870 20
-anchor c -fill red -text State
$canv create text 912 20
-anchor c -fill red -text Oper.
$canv create text 1080 20
-anchor c -fill red -text Address
$canv create text 1247 20
-anchor c -fill red -text Data

#State P2
for {set i 0} {$i<10} {incr i} {
$canv create rect 
850 [expr 30+15*$i] \ 
890 [expr 45+15*$i] \ 
-width 1 -fill black -outline #00FF00
$canv create text 870 [expr 37+15*$i] 
-anchor c -fill #00FF00 -text "" -tags StateP2($i)
}

#OPERATION P2
for {set i 0} {$i<10} {incr i} {
$canv create rect 
890 [expr 30+15*$i] 
935 [expr 45+15*$i] 
-width 1 -fill black -outline #00FF00
$canv create text 912 [expr 37+15*$i] 
-anchor c -fill #00FF00 -text "" -tags operationP2($i)
}

#ADDRESS P2
for {set i 0} {$i<10} {incr i} {
$canv create rect 
935 [expr 30+15*$i] 

APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

1225 \[expr 45+15*$i]\n-\width 1 -\fill black -\outline #00FF00
$canv create text 1080 \[expr 37+15*$i]\n-\anchor c -\text "" -\tags cpuaddrtextP2($i)
}

#Data P2
for {set i 0} {$i<10} {incr i} {
$canv create rect \
1225 \[expr 30+15*$i]\n1270 \[expr 45+15*$i]\n-\width 1 -\fill black -\outline #00FF00
$canv create text 1247 \[expr 37+15*$i]\n-\anchor c -\fill #00FF00 -\text "" -\tags DataCPU2($i)
}

#Horizontal CPU_REQ_ADDRESS
for {set i 0} {$i<32} {incr i} {
$canv create text \[expr 373+20*$i\] 200\n-\anchor c -\fill #00FF00 -\text \[expr 31-$i\] -\tags cpuaddrtext($i)
if {$i<29} {$canv create text \[expr 363+20*$i\] 200\n-\anchor c -\fill red -\text "|" -\tags cpuaddrtext1($i)
} else {$canv create text \[expr 363+20*$i\] 200\n-\anchor c -\fill blue -\text "|" -\tags cpuaddrtext1($i))
}$canv create text 1003 200\n-\anchor c -\fill blue -\text "|
for {set i 0} {$i<32} {incr i} {
$canv create rect \
[expr 363+20*$i] 210\n[expr 383+20*$i] 230\n-\width 1 -\fill black
$canv create text \[expr 373+20*$i\] 220\n
288
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
-geometry w595-h841

APPENDIX C TCL/TK CODE FOR CACHE CONTROLLER

-anchor c -fill #00FF00 -text "" -tags cpuaddrtext2($i)
}

for {set i 0} {$i<32} {incr i} {
$canv create text [expr 363+20*$i] 220
-anchor c -fill #00FF00 -text "|
}

#Tag Bits
$canv create rect \\
363 210 \\
942 230 \\
-outline red -width 2

#Index Bits
$canv create rect \\
944 210 \\
1003 230 \\
-outline blue -width 2

#Tag line & Arrows P0
$canv create line \\
370 230 \\
370 250 \\
-fill black -width 2 -tags lineP000
$canv create line \\
260 250 \\
370 250 \\
-fill black -width 2 -tags lineP001
$canv create line 260 250 260 290 -arrow last -fill black -width 2 -tags ar(P000)

#Index line & Arrows P0
$canv create line \\
964 230 \\
964 240 \\
-fill black -width 2 -tags lineP010
$canv create line \\
30 240 \\
964 240 \\
```

289
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```
# Cache P0
$canv create text 30 300  
-anchor c -fill #00FF00 -text Index
$canv create text 70 300  
-anchor c -fill #00FF00 -text Valid
$canv create text 110 300  
-anchor c -fill #00FF00 -text Dirty
$canv create text 260 300  
-anchor c -fill #00FF00 -text Tag
$canv create text 405 300  
-anchor c -fill #00FF00 -text Data

# Index
for {set i 0} {$i<8} {incr i} {
$canv create rect  
10 [expr 310+15*$i]  
50 [expr 325+15*$i]  
-width 1 -fill black -outline #00FF00
$canv create text 30 [expr 318+15*$i]  
-anchor c -text "" -tags Index0($i)
}

#Valid bit
```
for {set i 0} {i<8} {incr i} {
    $canv create rect \\
    50 [expr 310+15*$i]\\
    90 [expr 325+15*$i]\\
    -width 1 -fill black -outline #00FF00
    $canv create text 70 [expr 318+15*$i]\\
    -anchor c -fill #00FF00 -text "" -tags VBIT0($i)
}

#Dirty bit
for {set i 0} {i<8} {incr i} {
    $canv create rect \\
    90 [expr 310+15*$i]\\
    130 [expr 325+15*$i]\\
    -width 1 -fill black -outline #00FF00
    $canv create text 110 [expr 318+15*$i]\\
    -anchor c -fill #00FF00 -text "" -tags DBIT0($i)
}

#Tag 1 18 bits
for {set i 0} {i<8} {incr i} {
    $canv create rect \\
    130 [expr 310+15*$i]\\
    390 [expr 325+15*$i]\\
    -width 1 -fill black -outline #00FF00
    $canv create text 260 [expr 318+15*$i]\\
    -anchor c -text "" -tags Tag18bitstext0($i)
}

#Data 1 32 bits
for {set i 0} {i<8} {incr i} {
    $canv create rect \\
    390 [expr 310+15*$i]\\
    425 [expr 325+15*$i]\\
    -width 1 -fill black -outline #00FF00
    $canv create text 405 [expr 317+15*$i]\\
    -anchor c -fill #00FF00 -text "-8" -tags DataCM0($i)
}
#L2 Cache text
$canv create text 240 440
-anchor c -fill #00FF00 -text Private_L2_Cache_Memory_(Processor(0))

#Tag line & Arrows P1
$canv create line \
560 280 \n560 305 \n-fill blue -width 2 -tags lineP100
$canv create line \
560 305 \n685 305 \n-fill blue -width 2 -tags lineP101
$canv create line 685 230 685 290 -arrow last -fill black -width 2 -tags ar(P100)

#Index line & Arrows P1
$canv create line \
974 230 \n974 245 \n-fill black -width 2 -tags lineP110
$canv create line \
455 245 \n974 245 \n-fill black -width 2 -tags lineP111
$canv create line 455 245 455 290 -arrow last -fill black -width 2 -tags ar(P101)

# Cache P1
$canv create text 455 300\n-anchor c -fill #00FF00 -text Index
$canv create text 495 300\n-anchor c -fill #00FF00 -text Valid
$canv create text 535 300\n-anchor c -fill #00FF00 -text Dirty
$canv create text 685 300\n
-anchor c -fill #00FF00 -text Tag

$canv create text 825 300\n-anchor c -fill #00FF00 -text Data

#Index
for {set i 0} {$i<8} {incr i} {
  $canv create rect \n  430 [expr 310+15*$i]\n  480 [expr 325+15*$i]\n  -width 1 -fill black -outline #00FF00
  $canv create text 455 [expr 318+15*$i]\n  -anchor c -text "" -tags Index1($i)\n}

#Valid bit
for {set i 0} {$i<8} {incr i} {
  $canv create rect \n  480 [expr 310+15*$i]\n  510 [expr 325+15*$i]\n  -width 1 -fill black -outline #00FF00
  $canv create text 495 [expr 318+15*$i]\n  -anchor c -fill #00FF00 -text "" -tags VBIT1($i)\n}

#Dirty bit
for {set i 0} {$i<8} {incr i} {
  $canv create rect \n  510 [expr 310+15*$i]\n  550 [expr 325+15*$i]\n  -width 1 -fill black -outline #00FF00
  $canv create text 530 [expr 318+15*$i]\n  -anchor c -fill #00FF00 -text "" -tags DBIT1($i)\n}

APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

# Tag 1 18 bits
for {set i 0} {$i<8} {incr i} {
    $canv create rect \n    550 [expr 310+15*$i]\n    810 [expr 325+15*$i]\n    -width 1 -fill black -outline #00FF00
    $canv create text 680 [expr 318+15*$i]\n    -anchor c -text "" -tags Tag18bitstext1($i)
}

# Data 1 32 bits
for {set i 0} {$i<8} {incr i} {
    $canv create rect \n    810 [expr 310+15*$i]\n    845 [expr 325+15*$i]\n    -width 1 -fill black -outline #00FF00
    $canv create text 825 [expr 317+15*$i]\n    -anchor c -fill #00FF00 -text "-8" -tags DataCM1($i)
}

# L2 Cache text
$canv create text 665 440\n    -anchor c -fill #00FF00 -text Private_L2_Cache_Memory_(Processor(1))

# Tag line & Arrows P2
$canv create line \n    780 230 \n    780 255 \n    -fill black -width 2 -tags lineP200
$canv create line \n    780 255 \n    1105 255 \n    -fill black -width 2 -tags lineP201
$canv create line 1105 255 1105 290 -arrow last -fill black -width 2 -tags ar(P200)

# Index line & Arrows P2
$canv create line \n    984 230 \n
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

469  
470 984 265 \  
471 -fill black -width 2 -tags lineP210  
472 $canv create line \  
473 875 265 \  
474 984 265 \  
475 -fill black -width 2 -tags lineP211  
476 $canv create line 875 265 875 290 -arrow last -fill black -width 2 -tags ar(P201)  
477  
478 # Cache P2  
479 $canv create text 875 300\  
480 -anchor c -fill #00FF00 -text Index  
481 $canv create text 915 300\  
482 -anchor c -fill #00FF00 -text Valid  
483 $canv create text 955 300\  
484 -anchor c -fill #00FF00 -text Dirty  
485 $canv create text 1105 300\  
486 -anchor c -fill #00FF00 -text Tag  
487 $canv create text 1250 300\  
488 -anchor c -fill #00FF00 -text Data  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498 #Index  
499  
500 for {set i 0} {$i<8} {incr i} {  
501 $canv create rect \  
502 850 [expr 310+15*$i]\  
503 900 [expr 325+15*$i]\  
504 -width 1 -fill black -outline #00FF00  
505 $canv create text 875 [expr 318+15*$i]\  
506 -anchor c -text "$" -tags Index2($i)  
507 }  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
#Valid bit
for {set i 0} {$i<8} {incr i} {
    $canv create rect \ 900 [expr 310+15*$i]\ 930 [expr 325+15*$i]\ -width 1 -fill black -outline #00FF00
    $canv create text 915 [expr 318+15*$i]\ -anchor c -fill #00FF00 -text "" -tags VBIT2($i)
}

#Dirty bit
for {set i 0} {$i<8} {incr i} {
    $canv create rect \ 930 [expr 310+15*$i]\ 970 [expr 325+15*$i]\ -width 1 -fill black -outline #00FF00
    $canv create text 950 [expr 318+15*$i]\ -anchor c -fill #00FF00 -text "" -tags DBIT2($i)
}

#Tag 1 18 bits
for {set i 0} {$i<8} {incr i} {
    $canv create rect \ 970 [expr 310+15*$i]\ 1235 [expr 325+15*$i]\ -width 1 -fill black -outline #00FF00
    $canv create text 1103 [expr 317+15*$i]\ -anchor c -text "" -tags Tag18bitstext2($i)
}

#Data 1 32 bits
for {set i 0} {$i<8} {incr i} {
    $canv create rect \ 1235 [expr 310+15*$i]\ 1270 [expr 325+15*$i]\ -width 1 -fill black -outline #00FF00
    $canv create text 1250 [expr 317+15*$i]\ -anchor c -fill #00FF00 -text "-8" -tags DataCM2($i)
}
```

296
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
#L2 Cache text
$canv create text 1085 440
-anchor c -fill #00FF00 -text Private_L2_Cache_Memory_(Processor(2))

#Main Memory
$canv create text 900 470
-anchor c -fill #00FF00 -text Address
$canv create text 1075 470
-anchor c -fill #00FF00 -text Data

#Index
for {set i 0} {$i<16} {incr i} {
$canv create rect \
750 [expr 480+15*$i]\
1050 [expr 495+15*$i]\
-width 1 -fill black -outline #00FF00
$canv create text 900 [expr 488+15*$i]\
-anchor c -fill red -text "" -tags IndexMM($i)
}

#Data
for {set i 0} {$i<16} {incr i} {
$canv create rect \
1050 [expr 480+15*$i]\
1100 [expr 495+15*$i]\
-width 1 -fill black -outline #00FF00
$canv create text 1075 [expr 488+15*$i]\
-anchor c -fill red -text "" -tags DataMM($i)
$canv create text 1125 [expr 488+15*$i]\
-anchor c -fill red -text "" -tags DataMM2($i)
}

#Text Main Memory
$canv create text 950 730
-anchor c -fill #00FF00 -text Main_Memory
```

APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

# Timer
#$canv create text 80 30
-anchor c -fill #00FF00 -text Global_State -tags tmr

# Bullets Timeline
$canv create text 30 480 -text {State} -anchor c -fill #00FF00
$canv create text 40 520 -text {Address} -anchor c -fill #00FF00
$canv create text 30 565 -text {P0} -anchor c -fill #00FF00
$canv create text 30 605 -text {P1} -anchor c -fill #00FF00
$canv create text 30 645 -text {P2} -anchor c -fill #00FF00
$canv create text 40 685 -text {Memory} -anchor c -fill #00FF00

## Arc

#$canv create oval 80 80 87 87 -fill white
#$canv create arc 70 64 180 182 -outline #00FF00 -width 2 -style arc -start 45
#$canv create oval 160 80 167 87 -fill white
#$canv create arc 150 64 260 182 -outline #00FF00 -width 2 -style arc -start 45
#$canv create oval 240 80 247 87 -fill white
#$canv create arc 80 80 160 160 -outline white -width 2 -extent 180 -style arc
#P0
#$canv create oval 80 565 87 572 -fill white
#$canv create arc 70 549 163 667 -outline #00FF00 -width 2 -style arc -start 45
#$canv create oval 145 565 152 572 -fill white
#$canv create arc 135 549 228 667 -outline #00FF00 -width 2 -style arc -start 45
#$canv create oval 210 565 217 572 -fill white
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

#P1
$canv create oval 80 605 87 612 -fill white
$canv create arc 70 589 163 707 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 145 605 152 612 -fill white
$canv create arc 135 589 228 707 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 210 605 217 612 -fill white

#P2
$canv create oval 80 640 87 647 -fill white
$canv create arc 70 624 163 742 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 145 640 152 647 -fill white
$canv create arc 135 624 228 742 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 210 640 217 647 -fill white

#Memory
$canv create oval 80 680 87 687 -fill white
$canv create arc 70 664 163 782 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 145 680 152 687 -fill white
$canv create arc 135 664 228 782 -outline #00FF00 -width 2 -style arc -start 45
$canv create oval 210 680 217 687 -fill white

proc tmr {nl} {
}
variable canv;

set State [lindex $nl 0]
#$canv itemconfigure tmr -fill #00FF00 -text Global_State:$State

if {$State<9} {
$canv create text\[expr 83+65*$State\] 480\-fill #00FF00 -text $State
$canv create text\[expr 83+65*[expr $State+1]\] 480\-fill #00FF00 -text [expr $State+1]
} else {
$canv create text\[expr 83+65*$State\] 480\-fill #00FF00 -text $State
}

proc INDEX {nl} {
variable canv;

set IndexDecimal [lindex $nl 0]
set IndexBinary [lindex $nl 1]
$canv itemconfigure Index0($IndexDecimal) -fill #00FF00 -text $IndexBinary
$canv itemconfigure Index1($IndexDecimal) -fill #00FF00 -text $IndexBinary
$canv itemconfigure Index2($IndexDecimal) -fill #00FF00 -text $IndexBinary
}

proc CM {nl} {
variable canv;

set Pid [lindex $nl 0]
set Index [lindex $nl 1]
set Data [lindex $nl 2]
set Tag [lindex $nl 3]
set Index3 [lindex $nl 4]
set Index4 [lindex $nl 5]
set Index5 [lindex $nl 6]
set Index6 [lindex $nl 7]
set RW [lindex $nl 8]

if {$Pid==0} {
$canv itemconfigure lineP100 -fill black
$canv itemconfigure lineP101 -fill black
$canv itemconfigure ar(P100) -fill black
$canv itemconfigure lineP110 -fill black
$canv itemconfigure lineP111 -fill black
$canv itemconfigure ar(P101) -fill black
$canv itemconfigure lineP200 -fill black
$canv itemconfigure lineP201 -fill black
$canv itemconfigure ar(P200) -fill black
$canv itemconfigure lineP210 -fill black
$canv itemconfigure lineP211 -fill black
$canv itemconfigure ar(P201) -fill black
$canv itemconfigure lineP000 -fill red
$canv itemconfigure lineP001 -fill red
$canv itemconfigure ar(P000) -fill red
$canv itemconfigure lineP010 -fill blue
$canv itemconfigure lineP011 -fill blue
$canv itemconfigure ar(P001) -fill blue
$canv itemconfigure Index0($Index4) -fill white -text $Index3
if {$RW == 1} {
$canv itemconfigure DataCM0($Index4) -fill white -text $Data
}
$canv itemconfigure Tag18bitstext0($Index4) -fill white -text $Tag
} elseif {$Pid==1} {

APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
$canv itemconfigure lineP000 -fill black
$canv itemconfigure lineP001 -fill black
$canv itemconfigure ar(P000) -fill black
$canv itemconfigure lineP010 -fill black
$canv itemconfigure lineP011 -fill black
$canv itemconfigure ar(P001) -fill black
$canv itemconfigure lineP200 -fill black
$canv itemconfigure lineP201 -fill black
$canv itemconfigure ar(P200) -fill black
$canv itemconfigure lineP210 -fill black
$canv itemconfigure lineP211 -fill black
$canv itemconfigure ar(P201) -fill black
$canv itemconfigure lineP100 -fill red
$canv itemconfigure lineP101 -fill red
$canv itemconfigure ar(P100) -fill red
$canv itemconfigure lineP110 -fill blue
$canv itemconfigure lineP111 -fill blue
$canv itemconfigure ar(P101) -fill blue
$canv itemconfigure Index1($Index4) -fill white -text $Index3
if {$RW == 1} {
  $canv itemconfigure DataCM1($Index4) -fill white -text $Data
}
$canv itemconfigure Tag18bitstext1($Index4) -fill white -text $Tag
} elseif {$Pid==2} {
$canv itemconfigure lineP000 -fill black
$canv itemconfigure lineP001 -fill black
$canv itemconfigure ar(P000) -fill black
$canv itemconfigure lineP010 -fill black
$canv itemconfigure lineP011 -fill black
$canv itemconfigure ar(P001) -fill black
$canv itemconfigure lineP200 -fill black
$canv itemconfigure lineP201 -fill black
$canv itemconfigure ar(P200) -fill black
$canv itemconfigure lineP210 -fill black
$canv itemconfigure lineP211 -fill black
$canv itemconfigure ar(P201) -fill black
$canv itemconfigure lineP100 -fill red
$canv itemconfigure lineP101 -fill red
$canv itemconfigure ar(P100) -fill red
$canv itemconfigure lineP110 -fill blue
$canv itemconfigure lineP111 -fill blue
$canv itemconfigure ar(P101) -fill blue
$canv itemconfigure Index1($Index4) -fill white -text $Index3
  if ($RW == 1) {
    $canv itemconfigure DataCM1($Index4) -fill white -text $Data
  }
$canv itemconfigure Tag18bitstext1($Index4) -fill white -text $Tag
} elseif {$Pid==2} {
$canv itemconfigure lineP000 -fill black
$canv itemconfigure lineP001 -fill black
$canv itemconfigure ar(P000) -fill black
$canv itemconfigure lineP010 -fill black
$canv itemconfigure lineP011 -fill black
$canv itemconfigure ar(P001) -fill black
$canv itemconfigure lineP100 -fill black
$canv itemconfigure lineP101 -fill black
$canv itemconfigure ar(P100) -fill black
$canv itemconfigure lineP110 -fill black
$canv itemconfigure lineP111 -fill black
$canv itemconfigure ar(P101) -fill black
```
$canv itemconfigure lineP110 -fill black
$canv itemconfigure lineP111 -fill black
$canv itemconfigure ar(P101) -fill black
$canv itemconfigure lineP200 -fill red
$canv itemconfigure lineP201 -fill red
$canv itemconfigure ar(P200) -fill red
$canv itemconfigure lineP210 -fill blue
$canv itemconfigure lineP211 -fill blue
$canv itemconfigure ar(P201) -fill blue
$canv itemconfigure Index2($Index4) -fill white -text $Index3
if {$RW == 1} {
$canv itemconfigure DataCM2($Index4) -fill white -text $Data
$canv itemconfigure Tag18bitstext2($Index4) -fill white -text $Tag
}
proc IM {nl} {
variable canv;
set Data [lindex $nl 0]
set Addr [lindex $nl 1]
set IndexM [lindex $nl 2]
$canv itemconfigure DataMM($IndexM) -fill #00FF00 -text $Data
$canv itemconfigure IndexMM($IndexM) -fill #00FF00 -text $Addr
#$canv itemconfigure DataCM($IndexM) -fill red -text $Data
#$canv itemconfigure data0($Tick) -fill red -text $Data
}
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
proc MM {nl} {
    variable canv;

    set Data [lindex $nl 0]
    set Addr [lindex $nl 1]
    set IndexM [lindex $nl 2]
    set IndexC [lindex $nl 3]
    set Tick [lindex $nl 4]

    $canv itemconfigure DataMM2($IndexM) -fill white -text ""
    $canv itemconfigure DataMM2($IndexM) -fill white -text $Data
    $canv itemconfigure IndexMM($IndexM) -fill white -text $Addr
    #$canv itemconfigure DataCM($IndexM) -fill red -text $Data
    #$canv itemconfigure data0($Tick) -fill red -text $Data

    #$canv create oval\[expr 80+65*$Tick\] 680\[expr 87+65*$Tick\] 687\-fill #00FF00
    $canv create text\[expr 83+65*$IndexC\] 717\-fill red -text $Data
}

proc ABCD {nl} {
    variable canv;

    set Loc [lindex $nl 0]
    set Bits [lindex $nl 1]

    $canv itemconfigure cpuaddrtext2([expr ((Loc+31)+31)/(Loc+1)-Loc]) -fill white -text $Bits
```

proc CPUREQ {nl} {
    variable canv;

    set Pid [lindex $nl 0]
    set Index [lindex $nl 1]
    set Addr [lindex $nl 2]
    set RW [lindex $nl 3]
    set Data [lindex $nl 4]
    set State [lindex $nl 5]

    if {$Pid == 0} {
        #$canv itemconfigure rect1 -outline black
        #$canv itemconfigure rect2 -outline black
        #$canv itemconfigure rect0 -outline white
        $canv itemconfigure StateP0($State) -fill #00FF00 -text $State
    } elseif {$Pid == 1} {
        #$canv itemconfigure rect0 -outline black
        #$canv itemconfigure rect2 -outline black
        #$canv itemconfigure rect1 -outline white
        $canv itemconfigure StateP1($State) -fill #00FF00 -text $State
    }

    if {$RW==0} {
        $canv itemconfigure operationP0($Index) -fill #00FF00 -text $RW
    } elseif {$RW==1} {
        $canv itemconfigure operationP0($Index) -fill #00FF00 -text $RW
        $canv itemconfigure DataCPU0($Index) -fill white -text $Data
    }

    $canv itemconfigure cpuaddrtextP0($Index) -fill white -text $Addr
}

if {$Pid == 1} {
    #$canv itemconfigure rect0 -outline black
    #$canv itemconfigure rect2 -outline black
    #$canv itemconfigure rect1 -outline white
    $canv itemconfigure StateP1($State) -fill #00FF00 -text $State
}

if {$RW==0} {

$canv itemconfigure operationP1($Index) -fill #00FF00 -text $RW
} elseif {$RW==1} {
$canv itemconfigure operationP1($Index) -fill #00FF00 -text $RW
$canv itemconfigure DataCPU1($Index) -fill white -text $Data
}
$canv itemconfigure cpuaddrtextP1($Index) -fill white -text $Addr
} elseif {$Pid == 2} {
#canv itemconfigure rect0 -outline black
#canv itemconfigure rect1 -outline black
#canv itemconfigure rect2 -outline white
$canv itemconfigure StateP2($State) -fill #00FF00 -text $State
if {$RW==0} {
$canv itemconfigure operationP2($Index) -fill #00FF00 -text $RW
} elseif {$RW==1} {
$canv itemconfigure operationP2($Index) -fill #00FF00 -text $RW
$canv itemconfigure DataCPU2($Index) -fill white -text $Data
}
$canv itemconfigure cpuaddrtextP2($Index) -fill white -text $Addr
}
proc HM {nl} {
variable canv;
set PIDx [lindex $nl 0]
set PIDy [lindex $nl 1]
set PIDz [lindex $nl 2]
set stringx [lindex $nl 3]
set stringy [lindex $nl 4]
set stringz [lindex $nl 5]
set Index [lindex $nl 6]
set State [lindex $nl 7]
set VBITx [lindex $nl 8]
##set VBITy [lindex $nl 9]
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

#set VBITz [lindex $nl 10]
set DBITx [lindex $nl 9]
set DBITy [lindex $nl 10]
set DBITz [lindex $nl 11]
set Data [lindex $nl 12]
set Tick [lindex $nl 13]
set Memory [lindex $nl 14]
set Addr [lindex $nl 15]
set IndexM [lindex $nl 16]

#$canv itemconfigure VBIT($Index) -fill #00FF00 -text $VBIT
#$canv itemconfigure DBIT($Index) -fill #00FF00 -text $DBIT
#$canv itemconfigure DataMM($IndexM) -fill white -text $Memory
#$canv itemconfigure IndexMM($IndexM) -fill white -text $Addr

if {$PIDx==0} {
    $canv itemconfigure VBIT0($Index) -fill white -text $VBITx
    $canv itemconfigure DBIT0($Index) -fill white -text $DBITx
    if {$PIDy==1 && $PIDz==2} {
        #$canv itemconfigure VBIT1($Index) -fill white -text $VBITy
        $canv itemconfigure DBIT1($Index) -fill white -text $DBITy
        #$canv itemconfigure VBIT2($Index) -fill white -text $VBITz
        $canv itemconfigure DBIT2($Index) -fill white -text $DBITz
    } elseif {$PIDy==2 && $PIDz==1} {
        #$canv itemconfigure VBIT2($Index) -fill white -text $VBITy
        $canv itemconfigure DBIT2($Index) -fill white -text $DBITy
        #$canv itemconfigure VBIT1($Index) -fill white -text $VBITz
        $canv itemconfigure DBIT1($Index) -fill white -text $DBITz
    }

if {$stringx=="Write Miss"} {
    $canv itemconfigure DataCM0($Index) -fill white -text $Data
} elseif {$stringx=="Write Hit"} {
    $canv itemconfigure DataCM0($Index) -fill white -text $Data
} elseif {$stringx=="Read Miss"} {
    $canv itemconfigure DataCM0($Index) -fill white -text $Data
} elseif {$stringx=="Read Hit"} {
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
$canv itemconfigure DataCM0($Index) -fill white -text $Data
$canv itemconfigure DataCPU0($Tick) -fill white -text $Data
}
} elseif {$PIDx==1} {
$canv itemconfigure VBIT1($Index) -fill white -text $VBITx
$canv itemconfigure DBIT1($Index) -fill white -text $DBITx
if {$PIDy==0 && $PIDz==2} {
#canv itemconfigure VBIT0($Index) -fill white -text $VBITy
$canv itemconfigure DBIT0($Index) -fill white -text $DBITy
$canv itemconfigure VBIT2($Index) -fill white -text $VBITz
$canv itemconfigure DBIT2($Index) -fill white -text $DBITz
} elseif {$PIDy==2 && $PIDz==0} {
#canv itemconfigure VBIT2($Index) -fill white -text $VBITy
$canv itemconfigure DBIT2($Index) -fill white -text $DBITy
#canv itemconfigure VBIT0($Index) -fill white -text $VBITz
$canv itemconfigure DBIT0($Index) -fill white -text $DBITz
$canv itemconfigure DBIT0($Index) -fill white -text $DBITz
}
if {$stringx=="Write Miss"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
} elseif {$stringx=="Write Hit"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
} elseif {$stringx=="Read Miss"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
$canv itemconfigure DataCPU1($Tick) -fill white -text $Data
} elseif {$stringx=="Read Hit"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
$canv itemconfigure DataCPU1($Tick) -fill white -text $Data
}
} elseif {$PIDx==2} {
$canv itemconfigure VBIT2($Index) -fill white -text $VBITx
$canv itemconfigure DBIT2($Index) -fill white -text $DBITx
if {$PIDy==0 && $PIDz==1} {
#canv itemconfigure VBIT0($Index) -fill white -text $VBITy
$canv itemconfigure DBIT0($Index) -fill white -text $DBITy
#canv itemconfigure VBIT1($Index) -fill white -text $VBITz
$canv itemconfigure DBIT1($Index) -fill white -text $DBITz
} elseif {$PIDy==1 && $PIDz==0} {
#canv itemconfigure VBIT1($Index) -fill white -text $VBITy
$canv itemconfigure DBIT1($Index) -fill white -text $DBITy
$canv itemconfigure DBIT1($Index) -fill white -text $DBITz
}
if {$stringx=="Write Miss"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
} elseif {$stringx=="Write Hit"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
} elseif {$stringx=="Read Miss"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
$canv itemconfigure DataCPU1($Tick) -fill white -text $Data
} elseif {$stringx=="Read Hit"} {
$canv itemconfigure DataCM1($Index) -fill white -text $Data
$canv itemconfigure DataCPU1($Tick) -fill white -text $Data
}
}```
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
1015  #$canv itemconfigure VBIT0($Index) -fill white -text $VBITz
1016  $canv itemconfigure DBIT0($Index) -fill white -text $DBITz
1017  }
1018  if {$stringx=="Write Miss") {
1019  $canv itemconfigure DataCM2($Index) -fill white -text $Data
1020  } elseif {$stringx=="Write Hit") {
1021  $canv itemconfigure DataCM2($Index) -fill white -text $Data
1022  } elseif {$stringx=="Read Miss") {
1023  $canv itemconfigure DataCM2($Index) -fill white -text $Data
1024  $canv itemconfigure DataCPU2($Tick) -fill white -text $Data
1025  } elseif {$stringx=="Read Hit") {
1026  $canv itemconfigure DataCM2($Index) -fill white -text $Data
1027  $canv itemconfigure DataCPU2($Tick) -fill white -text $Data
1028  }
1029  }
1030
1031  #Processors' addresses values bullets
1032  if {$State==0) {
1033  if {$PIDx==0) {
1034  $canv create oval
1035  [expr 80+65*$State] 565
1036  [expr 87+65*$State] 572
1037  -fill white
1038  $canv create oval
1039  [expr 80+65*[expr $State+1]] 565
1040  [expr 87+65*[expr $State+1]] 572
1041  -fill white
1042  $canv create text
1043  [expr 83+65*$State] 582
1044  -fill white -text $Data
1045  $canv create arc [expr 80+65*$State-10] [expr 565-16]
1046  [expr 87+65*$State+76] [expr 572+95]
1047  -outline white -width 2 -style arc -start 45
1048  $canv create oval
1049  [expr 80+65*$State] 605
1050  [expr 87+65*$State] 612
1051  -fill #00FF00
1052  $canv create oval
```

309
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
1054  [expr 80+65*[$expr $State+1]] 605\n1055  [expr 87+65*[$expr $State+1]] 612\n1056  -fill #00FF00
1057  $canv create text
1058  [expr 83+65*$State] 622\n1059  -fill #00FF00 -text ""
1060  $canv create oval
1061  [expr 80+65*$State] 640\n1062  [expr 87+65*$State] 647\n1063  -fill #00FF00
1064  $canv create oval
1065  [expr 80+65*[$expr $State+1]] 640\n1066  [expr 87+65*[$expr $State+1]] 647\n1067  -fill #00FF00
1068  $canv create text
1069  [expr 83+66+$State] 657\n1070  -fill #00FF00 -text ""
1071  } elseif {$PIDx==1} {
1072  $canv create oval
1073  [expr 80+65+$State] 565\n1074  [expr 87+65+$State] 572\n1075  -fill #00FF00
1076  $canv create oval
1077  [expr 80+65*[$expr $State+1]] 565\n1078  [expr 87+65*[$expr $State+1]] 572\n1079  -fill #00FF00
1080  $canv create text
1081  [expr 83+65+$State] 582\n1082  -fill #00FF00 -text ""
1083  $canv create oval
1084  [expr 80+65+$State] 605\n1085  [expr 87+65+$State] 612\n1086  -fill white
1087  $canv create oval
1088  [expr 80+65*[$expr $State+1]] 605\n1089  [expr 87+65*[$expr $State+1]] 612\n1090  -fill white
1091  $canv create text
```

310
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
expr 83+65*$State] 622\n-false white -text $Data
$canv create arc [expr 80+65*$State-10] [expr 605-16]\nexpr 87+65*$State+76] [expr 612+95]\n-out outline white -width 2 -style arc -start 45
$canv create oval\nexpr 80+65*$State] 640\nexpr 87+65*$State] 647\n-false #00FF00
$canv create oval\nexpr 80+65*$State] [expr $State+1] 640\nexpr 87+65*$State+1] 647\n-false #00FF00
$canv create text\nexpr 83+65*$State] 657\n-false #00FF00 -text ""
} elseif {$PIDx==2} {
$canv create oval\nexpr 80+65*$State] 565\nexpr 87+65*$State] 572\n-false #00FF00
$canv create oval\nexpr 80+65*$State+1] 565\nexpr 87+65*$State+1] 572\n-false #00FF00
$canv create text\nexpr 83+65*$State] 582\n-false #00FF00 -text ""
$canv create oval\nexpr 80+65*$State] 605\nexpr 87+65*$State] 612\n-false #00FF00
$canv create oval\nexpr 80+65*$State+1] 605\nexpr 87+65*$State+1] 612\n-false #00FF00
$canv create text\nexpr 83+65*$State] 622
```

311
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
-fclose funky -text ""
$canv create oval
[expr 80+65*$State] 640
[expr 87+65*$State] 647
-fclose white
$canv create oval
[expr 80+65+[expr $State+1]] 640
[expr 87+65+[expr $State+1]] 647
-fclose white
$canv create text
[expr 83+65*$State] 657
-fclose white -text $Data
$canv create arc [expr 80+65*$State-10] [expr 640-16]
[expr 87+65*$State+76] [expr 647+95]
-fclose outline white -width 2 -style arc -start 45
}

) elseif {$State==9} {
if {$PIDx==0} {
$canv create oval
[expr 80+65*$State] 565
[expr 87+65*$State] 572
-fclose white
$canv create text
[expr 83+65*$State] 582
-fclose white -text $Data
$canv create text
[expr 83+65*$State] 582
-fclose white -text $Data

) elseif {$PIDx==1} {
$canv create oval
[expr 80+65*$State] 605
[expr 87+65*$State] 612
-fclose white
$canv create text
[expr 83+65*$State] 622
-fclose white -text $Data

) elseif {$PIDx==2} {
$canv create oval
```
```
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

[expr 80+65*$State] 640
[expr 87+65*$State] 647
-fill white
$canv create text
[expr 83+65*$State] 657
-fill white -text $Data
}
else {
if {$PIDx==0} {
$canv create oval
[expr 80+65*$State] 565
[expr 87+65*$State] 572
-fill white
$canv create oval
[expr 80+65*[expr $State+1]] 565
[expr 87+65*[expr $State+1]] 572
-fill white
$canv create text
[expr 83+65*$State] 582
-fill white -text $Data
$canv create arc [expr 80+65*$State-10] [expr 565-16]
[expr 87+65*$State+76] [expr 572+95]
-outline white -width 2 -style arc -start 45
$canv create oval
[expr 80+65*[expr $State+1]] 605
[expr 87+65*[expr $State+1]] 612
-fill #00FF00
$canv create text
[expr 83+65*$State] 622
-fill #00FF00 -text ""
$canv create oval
[expr 80+65*[expr $State+1]] 640
[expr 87+65*[expr $State+1]] 647
-fill #00FF00
$canv create text
[expr 83+65*$State] 657
-fill #00FF00 -text ""
} elseif {$PIDx==1} {
$canv create oval

APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
expr 80+65*[$State+1]
expr 87+65*[$State+1]
-f $00FF00
$canv create text
expr 83+65*$State
-f $00FF00 -text ""
$canv create oval
expr 80+65*$State
expr 87+65*$State
-f white
$canv create oval
expr 80+65*[$State+1]
expr 87+65*[$State+1]
-f white
$canv create oval
expr 83+65*$State
-f white -text $Data
$canv create oval
expr 80+65*[$State+1]
expr 87+65*[$State+1]
-f $00FF00
$canv create oval
expr 83+65*$State
-f $00FF00 -text ""

elseif ($PIDx==2) {
$canv create oval
expr 80+65*[$State+1]
expr 87+65*[$State+1]
-f $00FF00
$canv create oval
expr 83+65*$State
-f $00FF00 -text ""
$canv create oval
expr 80+65*[$State+1]
```

314
APPENDIX C. APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
expr 87+65*[expr $State+1]] 612
-f000
$canv create text
expr 83+65*$State] 622
-f000 -text ""
$canv create oval
expr 80+65*$State] 640
expr 87+65*$State] 647
-f000
$canv create oval
expr 80+65*[expr $State+1]] 640
expr 87+65*[expr $State+1]] 647
-f000
$canv create text
expr 83+65*$State] 657
-f000 -text $Data
$canv create oval
expr 80+65*$State] 680
expr 87+65*$State] 687
-f000
$canv create oval
expr 80+65*[expr $State+1]] 680
expr 87+65*[expr $State+1]] 687
-f000
$canv create text
expr 83+65*$State] 697
-f000 -text $Memory
$canv create arc [expr 80+65*$State+10] [expr 640+16]
expr 87+65*$State+76] [expr 647+95]
-outline white -width 2 -style arc -start 45
}
}

#Memory's address values bullets
if {($State<9) {
$canv create oval
expr 80+65*$State] 680
expr 87+65*$State] 687
-f000
$canv create oval
expr 80+65*[expr $State+1]] 680
expr 87+65*[expr $State+1]] 687
-f000
$canv create text
expr 83+65*$State] 697
-f000 -text $Memory
$canv create arc [expr 80+65*$State-10] [expr 680-16]
```
APPENDIX C: TCL/TK CODE FOR CACHE CONTROLLER

```tcl
expr 87+65*$State+76] [expr 687+95]\noutline white -width 2 -style arc -start 45
} else {
$canv create oval\n[expr 80+65*$State] 680\n[expr 87+65*$State] 687\n-fill white
$canv create text\n[expr 83+65*$State] 697\n-fill white -text $Memory
}

Addresses bullets
$canv create text\n[expr 83+65*$State] 520\n-fill white -text $Addr
}
```
Appendix D

Appendix D: Java Remote Method Invocation (RMI)
APPENDIX D. APPENDIX D: JAVA REMOTE METHOD INVOCATION (RMI)

Listing D.1: RMI Tempura Program

```c
/* -*- Mode: C -*-
 * This file is part Tempura: Interval Temporal Logic interpreter.
 * 
 * Copyright (C) 1998-2016 Nayef H.Alshammari, Antonio Cau
 * 
 * Tempura is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 * 
 * Tempura is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * 
 * You should have received a copy of the GNU General Public License
 * along with Tempura. If not, see <http://www.gnu.org/licenses/>.
 * 
 */

load "conversion".
load "exprog".

/* rmiregistry 0 */
/* rmiserver . RmiServerIntf RmiServer 1 */
/* rmiclient . RmiServerIntf RmiClient1 2 */
/* rmiclient . RmiServerIntf RmiClient2 3 */

define apidvar(X) = {X[0]}.
define apidval(X) = {X[1]}.
define avar1(X,a) = {X[a]}.
define aval1(X,b) = {X[b]}.
define atime1(X,c) = {strint(X[c])}.
define atime_micro1(X,d) = {X[d]}.
set print_states = true.

define get_var() = {
  exists T,Client,Data,Timestamp : {
    get2(T) and
    Client=strint(apidval(T)) and
    Data=strint(avall(T,3)) and
```
APPENDIX D. APPENDIX D: JAVA REMOTE METHOD INVOCATION (RMI)

```java
Timestamp =atime_micro1(T,4) and
format("Server is Receiving Assertion Data: X=%12d from Client %d at timestamp %s
", Data,Client,Timestamp) and empty
}
/* run */ define test() = {
exists v : {
    for v<2 do {get_var();skip}
}
}
```
define avar1(X,a) = {X[a]}.  
define avall(X,b) = {X[b]}.  
define atime(X,c) = {strint(X[c])}.  
define atime_micro(X,d) = {strint(X[d])}.  
set print_states = false.  
define assert(Client,Data,Timestamp) = {  
exists Client,Data,Timestamp : {  
format("\n") and  
format("Client %d is Sending %d to Interface\n",Client) and  
format("!PROG: assert Client:%d:X:%d:%s:!
",Client,Data,Timestamp)  
}  
}.  
define get_var() = {  
exists T : {  
get2(T) and  
Client =strint(apidval(T)) and  
Data =strint(avall(T,3)) and  
Timestamp =atime_micro(T,4) and  
format("Client %d is Receiving Assertion Data: Client=%d from %d from External Java ...
Program at
Timestamp=%s\n",Client,Client,Data,Timestamp) and assert(Client,Data,Timestamp) and empty  
}  
}.  
/* run */ define test_client1() = {skip and get_var()}.  

Listing D.3: Client 2 Tempura Program

/* -*- Mode: C -*-  
* This file is part Tempura: Interval Temporal Logic interpreter.  
*  
* Copyright (C) 1998-2016 Nayef H.Alshammari, Antonio Cau  
*  
* Tempura is free software: you can redistribute it and/or modify  
* it under the terms of the GNU General Public License as published by  
* the Free Software Foundation, either version 3 of the License, or  
* (at your option) any later version.  
*  
*/
* Tempura is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with Tempura. If not, see <http://www.gnu.org/licenses/>.
*
*/
load "conversion".
load "exporg".
/* rmiclient . RmiServerIntf RmiClient2 3 */
define apidvar(X) = {X[0]}.
define apidval(X) = {X[1]}.
define avar1(X,a) = {X[a]}.
define aval1(X,b) = {X[b]}.
define atime1(X,c) = {strint(X[c])}.
define atime_micro1(X,d) = {strint(X[d])}.
set print_states = false.
define assert(Client,Data,Timestamp) = {
exists Client,Data,Timestamp : {
    format("\n") and
    format("Client %d is Sending %d to Interface\n",Client) and
    format("!PROG: assert Client:%d:X:%d:%s:!
",Client,Data,Timestamp)
}
}.
define get_var() = {
exists T : {
    get2(T) and
    Client =strint(apidval(T)) and
    Data =strint(aval1(T,3)) and
    Timestamp =atime_micro1(T,4) and
    format("Client %d is Receiving Assertion Data: Client=%d from %d from External Java ...
    Program at
    Timestamp=%s\n",Client,Client,Data,Timestamp) and assert(Client,Data,Timestamp) and empty
}
}. /* run */
define test_client2() = {skip and get_var()}.
ANSWERS TO SELECTED EXERCISES

Listing D.4: Client 1 Java Program

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Date;
import java.util.Random;

public class RmiClient1 {
    private RmiClient1() {}
    public static void main(String args[]) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            RmiServerIntf stub = (RmiServerIntf) registry.lookup("RmiServerIntf");
            for(int i = 0; i < 1; i++) {
                Thread.sleep (1000);
                String Time = MicroTimestamp.INSTANCE.get();
                int id=1;
                Random rand = new Random();
                int X = rand.nextInt();
                System.out.println("!PROG: assert Client:"+id+:Data:"+X+":"+Time+":!");
                System.out.println("External Java Program is sending Assertion Data to Tempura Client="+id+" Data="+X+" Timestamp="+Time);
            }
        } catch(Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

public enum MicroTimestamp {
    INSTANCE ;
    private long startDate ;
    private long startNanoseconds ;
    private SimpleDateFormat dateFormat ;
    private MicroTimestamp() {
        this.startDate = System.currentTimeMillis() ;
        this.startNanoseconds = System.nanoTime() ;
        this.dateFormat = new SimpleDateFormat("HH-mm-ss-SSS") ;
    }
}
```
public String get() {
    long microSeconds = (System.nanoTime() - this.startNanoseconds) / 1000;
    long date = this.startDate + (microSeconds / 1000);
    return this.dateFormat.format(date) + String.format("%03d", microSeconds % 1000);
}

Listing D.5: Client 2 Java Program

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Date;
import java.util.Random;
public class RmiClient2 {
    private RmiClient2() {}
    public static void main(String args[]) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            RmiServerIntf stub = (RmiServerIntf) registry.lookup("RmiServerIntf");
            for(int i = 0; i < 1; i++) {
                Thread.sleep(1000);
                String Time = MicroTimestamp.INSTANCE.get();
                int id=2;
                Random rand = new Random();
                int X = rand.nextInt();
                System.out.println("!PROG: assert Client:" + id + ":Data:" + X + ":" + Time + ":!");
                System.out.println("External Java Program is sending Assertion Data to
                Tempura Client=" + id + ": Data=" + X + " Timestamp=" + Time);
            }
        } catch(Exception e) {
            System.err.println("Client exception: " + e.toString());
        }
    }
}
APPENDIX D. APPENDIX D: JAVA REMOTE METHOD INVOCATION (RMI)

```java
26     e.printStackTrace();
27  }
28  }
29  public enum MicroTimestamp {
30     INSTANCE;
31     private long startDate;
32     private long startNanoseconds;
33     private SimpleDateFormat dateFormat;
34     private MicroTimestamp() {
35         this.startDate = System.currentTimeMillis();
36         this.startNanoseconds = System.nanoTime();
37         this.dateFormat = new SimpleDateFormat("HH-mm-ss-SSS");
38     }
39     public String get() {
40         long microSeconds = (System.nanoTime() - this.startNanoseconds) / 1000;
41         long date = this.startDate + (microSeconds/1000);
42         return this.dateFormat.format(date) + String.format("%03d", microSeconds % 1000);
43     }
44  }
```

Listing D.6: Server Java Program

```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Date;
public class RmiServer implements RmiServerIntf {
    static Integer x = 0;
    public RmiServer() {}  
    public Integer getMessage() {
        return x++;
    }
}
```
public static void main(String args[]) {
    try {
        RmiServer obj = new RmiServer();
        RmiServerIntf stub = (RmiServerIntf) UnicastRemoteObject.exportObject(obj, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("RmiServerIntf", stub);
        System.err.println("Server ready");
    } catch (Exception e) {
        System.err.println("Server exception: ", e.toString());
        e.printStackTrace();
    }
}

Listing D.7: Server Interface Java Program

import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RmiServerIntf extends Remote {
    public Integer getMessage() throws RemoteException;
}
Appendix E

Appendix E: MATLAB Code for Correctness Properties
Listing E.1: MATLAB Code for Memory Consistency Property

```matlab
for c = [1 4 7 10 13 16 19 22 25 28]
    load AssertionData.txt
    State=AssertionData(c,1);
    Pid0=AssertionData(c,2);
    Pid1=AssertionData(c+1,2);
    Pid2=AssertionData(c+2,2);
    CacheP0=AssertionData(c,12);
    CacheP1=AssertionData(c+1,12);
    CacheP2=AssertionData(c+2,12);
    IndexP0=AssertionData(c,10);
    IndexP1=AssertionData(c+1,10);
    IndexP2=AssertionData(c+2,10);
    AddrMem=AssertionData(c,5);
    MemoryP0=AssertionData(c,13);
    MemoryP1=AssertionData(c+1,13);
    MemoryP2=AssertionData(c+2,13);
    switch c
        case 1
            f = 1;
        case 4
            f = 2;
        case 7
            f = 3;
        case 10
            f = 4;
        case 13
            f = 5;
        case 16
            f = 6;
        case 19
```

327
APPENDIX E. MATLAB CODE FOR CORRECTNESS PROPERTIES

```matlab
f = 7;
case 22
f = 8;
case 25
f = 9;
case 28
f = 10;
end

figure(f);
set(gcf, 'Position', [100, 100, 900, 700])
subplot(2,3,1);
plot(IndexP0,CacheP0,'b.','MarkerSize',20);
grid on
set(gca, 'XTick', 0:7)
xlim([-1 7])
ylim([-16 33])
xlabel('Index')
ylabel('Data')
legend(['Cache[',num2str(IndexP0) ']=' ,num2str(CacheP0)])
title("Cache of Pid "+Pid0+" at State "+State)

subplot(2,3,2);
plot(IndexP1,CacheP1,'g.','MarkerSize',20);
grid on
set(gca, 'XTick', 0:7)
xlim([-1 7])
ylim([-16 33])
xlabel('Index')
ylabel('Data')
legend(['Cache[',num2str(IndexP1) ']=' ,num2str(CacheP1)])
title("Cache of Pid "+Pid1+" at State "+State)

subplot(2,3,3);
plot(IndexP2,CacheP2,'r.','MarkerSize',20);
grid on
set(gca, 'XTick', 0:7)
xlim([-1 7])
ylim([-16 33])
xlabel('Index')
ylabel('Data')
legend(['Cache[',num2str(IndexP2) ']=' ,num2str(CacheP2)])
title("Cache of Pid "+Pid2+" at State "+State)
```

328
APPENDIX E. APPENDIX E: MATLAB CODE FOR CORRECTNESS PROPERTIES

85     xlabel('Index')
86     ylabel('Data')
87     legend(['Cache(',num2str(IndexP2),')=',num2str(CacheP2)])
88     title('Cache of Pid '+Pid2+' at State '+State)
89
90     subplot(2,3,[4,6]);
91     plot(AddrMem,MemoryP0,'m.','MarkerSize',20);
92     grid on
93     set(gca, 'XTick', 0:15)
94     xlim([-0.5 15])
95     ylim([-16 33])
96     xlabel('Address')
97     ylabel('Data')
98     legend(['Memory(',num2str(AddrMem),')=',num2str(MemoryP0)])
99     title('Main Memory at State '+State)
100
101 end
102
disp('Correctness Property 1: Memory Consistency is Done!')

Listing E.2: MATLAB Code for Cache Coherence Property

104 for c = [1 4 7 10 13 16 19 22 25 28]
105     load AssertionData.txt
106     State=AssertionData(c,1);
107     Pid0=AssertionData(c,2);
108     Pid1=AssertionData(c+1,2);
109     Pid2=AssertionData(c+2,2);
110     IndexP0=AssertionData(c,10);
111     IndexP1=AssertionData(c+1,10);
112     IndexP2=AssertionData(c+2,10);
113     MSIP0=AssertionData(c,14);
114     MSIP1=AssertionData(c+1,14);
APPENDIX E: MATLAB CODE FOR CORRECTNESS PROPERTIES

MSIP2 = AssertionData(c+2,14);

switch c
    case 1
        f = 1;
    case 4
        f = 2;
    case 7
        f = 3;
    case 10
        f = 4;
    case 13
        f = 5;
    case 16
        f = 6;
    case 19
        f = 7;
    case 22
        f = 8;
    case 25
        f = 9;
    case 28
        f = 10;
end

figure(f);
    set(gca, 'Position', [100, 100, 900, 700])
    subplot(3,1,1);
    C = [0 0 0];
    if MSIP0 == 1
        C = [1 0 0];
    elseif MSIP0 == 2
        C = [0 0 1];
    else
        C = [0 1 0];
    end

    plot(IndexP0,MSIP0,'color',C,'marker','.','MarkerSize',20);
    grid on
APPENDIX E. APPENDIX E: MATLAB CODE FOR CORRECTNESS PROPERTIES

```matlab
set(gca, 'XTick', 0:7)
set(gca, 'YTick', 0:4)
xlim([-1 7])
ylim([0 3.5])
xlabel('Index')
ylabel('MSI')
if MSIP0 == 1
  legend('Modified');
elseif MSIP0 == 2
  legend('Shared');
else
  legend('Invalid');
end
title("Coherence State of Cache Blocks of Processor "+Pid0+" at State "+State)

subplot(3,1,2);
C = [0 0 0];
if MSIP1 == 1
  C = [1 0 0];
elseif MSIP1 == 2
  C = [0 0 1];
else
  C = [0 1 0];
end
plot(IndexP1,MSIP1,'color',C,'marker','.','MarkerSize',20);
grid on
set(gca, 'XTick', 0:7)
set(gca, 'YTick', 0:4)
xlim([-1 7])
ylim([0 3.5])
xlabel('Index')
ylabel('MSI')
if MSIP1 == 1
  legend('Modified');
elseif MSIP1 == 2
  legend('Shared');
else
  legend('Invalid');
end
```

331
APPENDIX E. APPENDIX E: MATLAB CODE FOR CORRECTNESS PROPERTIES

end

disp('Correctness Property 2: Cache Coherence is Done!')