Formal Verification of an OCCAM-to-FPGA Compiler and its Generated Logic Circuits

D. Pizarro De La Iglesia
Submitted for the Degree of Doctor of Philosophy from the University of Surrey

Department of Computing
Faculty of Engineering and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

July 2009

© D. Pizarro De La Iglesia 2009
Abstract

As custom logic circuits (e.g. field-programmable gate arrays, or FPGAs) have become larger, the limitations of conventional design flows have become more apparent. For large designs, verification by simulation is now impractical.

One solution to this challenge is hardware/software co-design, in which the desired logic is specified using a high-level, or programming, approach. Here, the requirements are to incorporate the parallelism from the original specification, as well as to accommodate greater complexity through various abstraction techniques. Verification may be achieved by simulation at an algorithmic level, or by formal proofs. However, verification by a particular circuit runs into difficulties of state-space explosion in a state-based model-checking scheme.

The research that is reported in this thesis has addressed the formal verification of a compiler that generates FPGA circuits. Rather than proving the correctness of every circuit generated by a compiler, the approach has been to validate the behaviour of the compiler itself. The source language that has been used is a variant of OCCAM, which incorporates fine-grained parallelism and message-passing along channels between parallel processes. The syntax of an OCCAM program can be mapped onto an abstract syntax tree, and each component of this tree can then be mapped into hardware. It is then necessary to prove that the hardware for each component correctly implements its specification, which can be performed by specifying each in CSP and model-checking them to check for equivalence. The components must also be proved to compose correctly, so that they may be plugged together in any way that satisfies the syntax of the source language. Finally, the correct composition of components is enforced by the type-checking of the compiler itself, using Java classes and inheritance to ensure that the CSP for each of the components being checked is manufactured within the same Java methods that create the hardware circuits.

The thesis discusses the proof strategy and shows how correctness can be justified. It concludes with some examples of simple OCCAM programs that have been translated into hardware and executed on real FPGA devices, correctly at the first attempt.
Contents

Abstract ................................................................................................................................ ii

Contents ................................................................................................................................ iii

List of Figures ................................................................................................................... xvi

List of Tables ................................................................................................................... xxv

Acknowledgements ......................................................................................................... xxvi

1 Introduction ....................................................................................................................... 1

1.1 Thesis Contributions ................................................................................................ 2

1.2 Possible Utilisation .................................................................................................. 3

1.3 Thesis Outline .......................................................................................................... 4

2 Literature Review .............................................................................................................. 8

2.1 FPGA ....................................................................................................................... 8

2.1.1 Programming FPGAs ......................................................................................... 8

2.2 CSP ........................................................................................................................ 8

2.2.1 Processes and Events ......................................................................................... 8

2.2.2 External Choice: Deterministic Choice ............................................................. 8

2.2.3 Internal Choice: Non-deterministic Choice ....................................................... 8

2.2.4 Hiding ................................................................................................................ 8

2.2.5 Boolean Guards .................................................................................................. 8

2.2.6 Let Within .......................................................................................................... 8

2.2.7 Alphabetised Parallel ......................................................................................... 8

2.2.8 Shared Parallel ................................................................................................... 8

2.2.9 Interleaving ........................................................................................................ 8

2.2.10 Congruence & Monotonicity ............................................................................. 8

2.3 FDR ........................................................................................................................ 8
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>OCCAM</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Clocked Logic</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>The IO-PAR Design Paradigm</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>SableCC</td>
<td>8</td>
</tr>
<tr>
<td>2.8</td>
<td>Comparison with Related Work</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Preliminary Work: Automatic CSP Modelling of Generated Logic Circuits</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Modelling Logic Circuits</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Modelling Logic Components</td>
<td>8</td>
</tr>
<tr>
<td>3.3</td>
<td>Why Use CSP?</td>
<td>8</td>
</tr>
<tr>
<td>3.4</td>
<td>Component Models in CSP</td>
<td>8</td>
</tr>
<tr>
<td>3.5</td>
<td>Circuit Models in CSP</td>
<td>8</td>
</tr>
<tr>
<td>3.6</td>
<td>Proving the Logic Circuit is Equivalent to a Low Level Specification</td>
<td>8</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Performance Problems: State Space Explosion</td>
<td>8</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Utilising CHASE to Achieve Viable Runtime Simulations</td>
<td>8</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Implementation &amp; Model Simplification</td>
<td>8</td>
</tr>
<tr>
<td>3.7</td>
<td>Experimental Results</td>
<td>8</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusion</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Main Work</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Premise for Work</td>
<td>8</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Implementation Implications of Premise</td>
<td>8</td>
</tr>
<tr>
<td>4.2</td>
<td>Assumptions made within the Compiler and Proof Strategy</td>
<td>8</td>
</tr>
<tr>
<td>4.3</td>
<td>Project Overview</td>
<td>8</td>
</tr>
<tr>
<td>4.4</td>
<td>Logic Component Representation</td>
<td>8</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Why Continue Using CSP?</td>
<td>8</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Design Considerations &amp; Implications</td>
<td>8</td>
</tr>
<tr>
<td>4.4.2.1</td>
<td>Combinatorial Logic</td>
<td>8</td>
</tr>
</tbody>
</table>
4.4.2.2 Clocked Logic
4.4.2.3 Adapting IO-PAR to OI-SEQ
4.5 Proof Framework
4.5.1 Appendix D Overview – A Single-Type Super-Type Component
4.5.2 Appendix E Overview – A Single-Type Implemented Component
4.5.3 Alternative Single-Type Component Models
4.5.4 Special Multi Type Components
4.6 Proof/Compiler Integration
5 Results
5.1 Example 1: Commsime – Version 1
5.2 Example 2: Commsime – Version 2
5.3 Example 3: A Digital Clock Displayed on an LCD
5.3.1 Main
5.3.2 LCD_PowerOnInit
5.3.3 LCD_Nibble
5.3.4 LCD_Delay
5.3.5 LCD_DisplayTime
5.3.6 DigitalClock
5.3.7 DebounceHour and DebounceMin
5.3.8 Compiling to Hardware
6 Future Work & Conclusion
Appendix A Other Preliminary Work
A.1 Occam Grammar Modifications
A.2 AST Transformer Extensions
A.3 Graphical Circuit Visualiser
A.3.1 Auto Layout Simplification
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2.13</td>
<td>GenSpec Assertions 13: GenSpec 4 With Signals Hidden is Similar to</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>GenSpec 6</td>
<td></td>
</tr>
<tr>
<td>D.3</td>
<td>Conclusion &amp; Evaluation</td>
<td>8</td>
</tr>
<tr>
<td>D.4</td>
<td>Future Work</td>
<td>8</td>
</tr>
<tr>
<td>D.4.1</td>
<td>Linking Clock Cycle Annotations to a Higher Level Specification</td>
<td>8</td>
</tr>
<tr>
<td>D.4.1.1</td>
<td>GenSpec 7: Software Component Generic Spec Model</td>
<td>8</td>
</tr>
<tr>
<td>D.4.1.2</td>
<td>GenSpec Assertions 14: Linking GenSpec 6 to GenSpec 7</td>
<td>8</td>
</tr>
<tr>
<td>D.4.2</td>
<td>Model Simplifications</td>
<td>8</td>
</tr>
<tr>
<td>Appendix E</td>
<td>Single Type Component Implemented Model Example</td>
<td>8</td>
</tr>
<tr>
<td>E.1</td>
<td>Models &amp; Specifications</td>
<td>8</td>
</tr>
<tr>
<td>E.1.1</td>
<td>ImpSpec 1: Valid Low Level Behaviour</td>
<td>8</td>
</tr>
<tr>
<td>E.1.2</td>
<td>ImpSpec 2: Low Level Behaviour with Explicit Deadlocking</td>
<td>8</td>
</tr>
<tr>
<td>E.1.3</td>
<td>ImpSpec 3: Model of Implemented Logic</td>
<td>8</td>
</tr>
<tr>
<td>E.1.4</td>
<td>ImpSpec 4: Annotation Only Specification</td>
<td>8</td>
</tr>
<tr>
<td>E.2</td>
<td>Assertions: Linking the Models Together</td>
<td>8</td>
</tr>
<tr>
<td>E.2.1</td>
<td>ImpSpec Assertion 1: Deadlock Freedom</td>
<td>8</td>
</tr>
<tr>
<td>E.2.2</td>
<td>ImpSpec Assertion 2: Super Type Control Only Limits the Behaviour</td>
<td>8</td>
</tr>
<tr>
<td>E.2.3</td>
<td>ImpSpec Assertion 3: Expected 'IF' Component Behaviour is Deadlock-Free</td>
<td>8</td>
</tr>
<tr>
<td>E.2.4</td>
<td>ImpSpec Assertion 4: Expected Boundary Behaviour Refines Super Type</td>
<td>8</td>
</tr>
<tr>
<td>E.2.5</td>
<td>ImpSpec Assertions 5: Correctly Driven Implementation Behaves as Expected</td>
<td>8</td>
</tr>
<tr>
<td>E.2.6</td>
<td>ImpSpec Assertion 6: Annotation Outer Level Does Not Introduce Deadlock</td>
<td>8</td>
</tr>
<tr>
<td>E.2.7</td>
<td>ImpSpec Assertion 7: Expected High Level Behaviour is Deadlock-Free</td>
<td>8</td>
</tr>
<tr>
<td>E.2.8</td>
<td>ImpSpec Assertions 8: Component Behaves Similarly to Expected Higher Spec</td>
<td>8</td>
</tr>
<tr>
<td>E.3</td>
<td>Conclusions &amp; Evaluation</td>
<td>8</td>
</tr>
<tr>
<td>E.4</td>
<td>Future Work</td>
<td>8</td>
</tr>
</tbody>
</table>
E.4.1 Linking Clock Cycle Annotations to Higher Specification ....................................... 8
E.4.1.1 ImpSpec 5: Software Specification Model ............................................................ 8
E.4.1.2 ImpSpec Assertion 9: Software Specification is a Refinement of Super Type ......................... 8
Appendix F Multi-Type Component Generic Specification Model Example ................ 8
F.1 Models & Specifications .......................................................................................... 8
  F.1.1 GenSpec 1: Valid Low Level Behaviour ............................................................... 8
  F.1.2 GenSpec 2: Low Level Behaviour with Explicit Deadlocking ............................. 8
  F.1.3 GenSpec 3: Correct Component Driving .............................................................. 8
  F.1.4 GenSpec 4: Annotated Low Level Behaviour with Explicit Deadlocking ............ 8
  F.1.5 GenSpec 5: Annotating the Outer Layer .............................................................. 8
  F.1.6 GenSpec 6: Clock Cycle Higher Generic Specification ......................................... 8
F.2 Assertions: Linking the Models Together ................................................................ 8
  F.2.1 GenSpec Assertion 1: Initial Deadlock-Free Check ............................................. 8
  F.2.2 GenSpec Assertion 2: GenSpec 2 Contains GenSpec 1 Behaviour ....................... 8
  F.2.3 GenSpec Assertion 3: GenSpec 3 Compatible with GenSpec 2 ............................ 8
  F.2.4 GenSpec Assertion 4: GenSpec 3 Removes Deadlock from GenSpec 2 ............... 8
  F.2.5 GenSpec Assertions 5: GenSpec 3 Removes Only Incorrect Driving ................. 8
  F.2.6 GenSpec Assertions 6: Properties of the Annotation Events ............................... 8
  F.2.7 GenSpec Assertion 7: GenSpec 3 Compatible with GenSpec 4 ............................ 8
  F.2.8 GenSpec Assertion 8: GenSpec 3 Removes Deadlock From GenSpec 4 ............... 8
  F.2.9 GenSpec Assertions 9: GenSpec 4 is an Annotated GenSpec 2 ............................ 8
  F.2.10 GenSpec Assertion 10: GenSpec 6 Deadlock-Free .............................................. 8
  F.2.11 GenSpec Assertions 11: GenSpec 5 Annotates Outer Level Correctly ................. 8
  F.2.12 GenSpec Assertions 12: GenSpec 5 Does Not Introduce Extra Behaviour ............ 8
  F.2.13 GenSpec Assertions 13: GenSpec 4 With Signals Hidden is Similar to GenSpec 6 ...
## Contents

I.1 Models & Specifications

1.1.1 ImpSpec 1: Valid Low Level Behaviour

1.1.2 ImpSpec 2: Low Level Behaviour with Explicit Deadlocking

1.1.3 ImpSpec 3: Model of Implemented Logic

1.1.4 ImpSpec 4: Annotation Only Specification

1.1.5 ImpSpec 5: Software Specification Model

I.2 Assertions: Linking the Models Together

1.2.1 ImpSpec Assertion 1: Deadlock-Free

1.2.2 ImpSpec Assertion 2: Super Type Control Only Limits the Behaviour

1.2.3 ImpSpec Assertion 3: Expected ‘IF’ Component Behaviour is Deadlock-Free

1.2.4 ImpSpec Assertion 4: Expected Boundary Behaviour Refines Super Type

1.2.5 ImpSpec Assertions 5: Correctly Driven Implementation Behaves as Expected

1.2.6 ImpSpec Assertion 6: Annotation Outer Level Does Not Introduce Deadlock

1.2.7 ImpSpec Assertion 7: Expected High Level Behaviour is Deadlock-Free

1.2.8 ImpSpec Assertions 8: Component Behaves Similarly to Expected Higher Spec

1.2.9 ImpSpec Assertion 9: Software Specification is a Refinement of Super Type

Appendix J Pattern Matched Selective Renaming Alternative

J.1 Methodology

J.2 Conclusion

Appendix K Simplified CSP Models Without Reset Feature

K.1 Logic Components

K.2 Super Type Specifications

K.2.1 Control Flow Process

K.2.1.1 Control Flow Process: Correct Control

K.2.1.2 Control Flow Process: Low Level Specification
List of Figures

Figure 1: How the CSP proof fits with the hardware compilation ........................................ 1
Figure 2: Pictorial representation of the Logic Circuit from Figure 5 on page 25 ............... 8
Figure 3: Simplified D-Type Flip-Flop Model ..................................................................... 8
Figure 4: An Instantiation of a Two Input AND Gate ......................................................... 8
Figure 5: Example CSP of Running Logic Components in Parallel .................................... 8
Figure 6: Algorithmic Specification of the Logic Circuit from Figure 5 ............................ 8
Figure 7: Simple Testing of the Generated Circuit .............................................................. 8
Figure 8: Initial T-Type Flip-Flop Model Developed .......................................................... 8
Figure 9 CSP specification of a 4-bit counter process ......................................................... 8
Figure 10 Initial OCCAM counter program ........................................................................ 8
Figure 11 Single-value commstime program ...................................................................... 8
Figure 12: Triple-value commstime program ...................................................................... 8
Figure 13: Graphical representation of an "IF THEN ELSE" component ............................ 8
Figure 14: Simplified class diagram of a subset of components and their super types ...... 8
Figure 15: Overview of CSP proof structure for a single component ................................. 8
Figure 16: CSP model of D-Type Flip-Flop used in chapter 3 ............................................ 8
Figure 17: An Instantiation of a Two Input AND Gate ....................................................... 8
Figure 18: AND Gate CSP Model: IO-SEQ ...................................................................... 8
Figure 19: An Instantiation of a D Type Flip Flop ............................................................... 8
Figure 20: D-Type Flip-Flop CSP Model: IO-PAR ............................................................. 8
Figure 21: A simple CSP definition of an OI-SEQ component .......................................... 8
Figure 22: A CSP definition of an OI-SEQ component using channels .............................. 8
Figure 23: IO-SEQ and IO-PAR connected in a loop ....................................................... 8
Figure 24: OI-SEQ replaces IO-PAR in IO-SEQ/IO-PAR loop ....................................... 8
Figure 47: A pictorial example of a three input AND gate .................................................. 8
Figure 48: A pictorial example of a three input OR gate ..................................................... 8
Figure 49: CSP Definition of a D-Type Flip-Flop from [Peel & Wong, 2004] ................... 8
Figure 50: CSP Definition of a D-Type Flip-Flop from Appendix B, used in Chapter 3 .. 8
Figure 51: A Modified CSP Definition of a D-Type Flip-Flop – Version 1 ....................... 8
Figure 52: Overview of the relationship between the numerous models of an OCCAM supertype ....................................................................................................................... 8
Figure 53: Low Level Generic Control Flow Specification ................................................ 8
Figure 54: Low Level Generic Control Flow Specification with Explicit Deadlocking .... 8
Figure 55: Generic Control Flow Specification - Correct Driving Limiter ....................... 8
Figure 56: Annotated Generic Control Flow Specification with Explicit Deadlocking ..... 8
Figure 57: Generic Control Flow Annotate Outer Layer ..................................................... 8
Figure 58: Generic Control Flow Annotation Specification ................................................ 8
Figure 59: Example of GenSpec Assertion 1 for Control Flow Process ......................... 8
Figure 60: Example of GenSpec Assertion 2 for Control Flow Process ......................... 8
Figure 61: Example of GenSpec Assertion 3 for Control Flow Process ......................... 8
Figure 62: Example of GenSpec Assertion 4 for Control Flow Process ......................... 8
Figure 63: Example of GenSpec Assertions 5 for Control Flow Process ......................... 8
Figure 64: Example of GenSpec Assertions 6 for Control Flow Process ......................... 8
Figure 65: Example of GenSpec Assertion 7 for Control Flow Process ......................... 8
Figure 66: Example of GenSpec Assertion 8 for Control Flow Process ......................... 8
Figure 67: Example of GenSpec Assertions 9 for Control Flow Process ......................... 8
Figure 68: Example of GenSpec Assertion 10 for Control Flow Process ....................... 8
Figure 69: Example of GenSpec Assertion 11 for Control Flow Process ....................... 8
Figure 70: Example of GenSpec Assertions 12 for Control Flow Process ....................... 8
Figure 71: Example of GenSpec Assertions 13 for Control Flow Process ....................... 8
Figure 72: Example of Simplified Control Flow Annotation Specification ........................................ 8
Figure 73: Example of how to link Annotation Spec to Simplified Spec ........................................ 8
Figure 74: Overview of proof structure for an implemented component ........................................ 8
Figure 75: Low Level 'IF' Component Desired Specification ....................................................... 8
Figure 76: Low Level 'IF' Component Generic Specification with Explicit Deadlocking .......... 8
Figure 77: CSP Model of the Logic Circuit Segment of the Implemented 'IF' Component ............... 8
Figure 78: A Graphical Depiction of the Logic Segment for an 'IF' Component ......................... 8
Figure 79: IF Component Annotation Only Specification .......................................................... 8
Figure 80: 'IF' Component Deadlock-Free Assertion ................................................................. 8
Figure 81: Super Type Component Limits the Behaviour of the Implementation ......................... 8
Figure 82: Expected 'IF' Component Behaviour is Deadlock-Free ............................................. 8
Figure 83: Expected 'IF' Component Behaviour is a Refinement of Super Type ........................... 8
Figure 84: Correctly Driven 'IF' Component Behaves as Expected ........................................... 8
Figure 85: Annotating outer level of 'IF' component does not introduce deadlock ..................... 8
Figure 86: 'IF' Component High Level Behaviour is Deadlock-Free .......................................... 8
Figure 87: 'IF' Component Behaves Similarly to Expected Higher Behaviour ............................. 8
Figure 88: 'IF' Component Software Annotation Behavioural Specification ................................ 8
Figure 89: Higher Software Specification is a Refinement of the Super Type ............................. 8
Figure 90: Low Level Generic Data Storage Specification .......................................................... 8
Figure 91: Low Level Generic Data Storage Specification with Explicit Deadlocking ............... 8
Figure 92: Generic Control Data Storage Specification - Correct Driving Limiter ....................... 8
Figure 93: Annotated Generic Data Storage Specification with Explicit Deadlocking ................. 8
Figure 94: Generic Data Storage Annotate Outer Layer ............................................................. 8
Figure 95: Generic Data Storage Annotation Specification ....................................................... 8
Figure 96: Example of GenSpec Assertion 1 for Data Storage Specification ............................. 8
Figure 97: Example of GenSpec Assertion 2 for Data Storage Specification
Figure 98: Example of GenSpec Assertion 3 for Data Storage Specification
Figure 99: Example of GenSpec Assertion 4 for Data Storage Specification
Figure 100: Example of GenSpec Assertions 5 for Data Storage Specification
Figure 101: Example of GenSpec Assertions 6 for Data Storage Specification
Figure 102: Example of GenSpec Assertion 7 for Data Storage Specification
Figure 103: Example of GenSpec Assertion 8 for Data Storage Specification
Figure 104: Example of GenSpec Assertions 9 for Data Storage Specification
Figure 105: Example of GenSpec Assertion 10 for Data Storage Specification
Figure 106: Example of GenSpec Assertion 11 for Data Storage Specification
Figure 107: Example of GenSpec Assertions 12 for Data Storage Specification
Figure 108: Example of GenSpec Assertions 13 for Data Storage Specification
Figure 109: Assertions Demonstrating Similarities with Single Type Generic Interface Specifications
Figure 110: Low Level 'FlipFlopStorage' Component Desired Specification
Figure 111: Low Level 'FlipFlopStorage' Component Generic Specification with Explicit Deadlocking
Figure 112: CSP Model of the Logic Circuit Segment of the 'FlipFlopStorage' Component
Figure 113: FlipFlopStorage Component Annotation Only Specification
Figure 114: 'FlipFlopStorage' Component Deadlock-Free Assertion
Figure 115: Super Type Component Limits the Behaviour of the Implementation
Figure 116: Expected Component Behaviour is Deadlock-Free
Figure 117: Expected Component Behaviour is a Refinement of Super Type
Figure 118: Correctly Driven Component Behaves as Expected
Figure 119: Annotating outer level of the component does not introduce deadlock
Figure 120: Components High Level Behaviour is Deadlock-Free
Figure 121: Component Behaves Similarly to Expected Higher Behaviour....................... 8
Figure 122: Low Level Generic Control Flow Specification ........................................... 8
Figure 123: Low Level Generic Control Flow Specification with Explicit Deadlocking... 8
Figure 124: Generic Control Flow Specification - Correct Signal Driver...................... 8
Figure 125: Annotated Generic Control Flow Specification with Explicit Deadlocking.... 8
Figure 126: Generic Control Flow Annotate Outer Layer............................................. 8
Figure 127: Generic Control Flow Annotation Specification......................................... 8
Figure 128: Control Generic Control Flow Annotation Specification.............................. 8
Figure 129: Annotate Generic Control Flow Annotation Specification........................... 8
Figure 130: Example of Simplified Control Flow Annotation Specification.................. 8
Figure 131: Control Process for Annotation Only Control Flow Specification .............. 8
Figure 132: Example of GenSpec Assertion 1 for Control Flow Process....................... 8
Figure 133: Example of GenSpec Assertion 2 for Control Flow Process....................... 8
Figure 134: Example of GenSpec Assertion 3 for Control Flow Process....................... 8
Figure 135: Example of GenSpec Assertion 4 for Control Flow Process....................... 8
Figure 136: Example of GenSpec Assertions 5 for Control Flow Process...................... 8
Figure 137: Example of GenSpec Assertions 6 for Control Flow Process...................... 8
Figure 138: Example of GenSpec Assertion 7 for Control Flow Process....................... 8
Figure 139: Example of GenSpec Assertion 8 for Control Flow Process....................... 8
Figure 140: Example of GenSpec Assertions 9 for Control Flow Process...................... 8
Figure 141: Example of GenSpec Assertion 10 for Control Flow Process...................... 8
Figure 142: Example of GenSpec Assertion 11 for Control Flow Process...................... 8
Figure 143: Example of GenSpec Assertions 12 for Control Flow Process...................... 8
Figure 144: Example of GenSpec Assertions 13 for Control Flow Process...................... 8
Figure 145: Linking GenSpec 10 to GenSpec 5 ............................................................. 8
Figure 146: Example of how to link Annotation Spec to Simplified Spec..................... 8
Figure 147: Low Level 'IF' Component Desired Specification ........................................... 8
Figure 148: Low Level 'IF' Component Generic Specification with Explicit Deadlocking 8
Figure 149: CSP Model of the Logic Circuit Segment of the Implemented 'IF'
Component.................................................................................................................... 8
Figure 150: A Graphical Depiction of the Logic Segment for an 'IF' Component......... 8
Figure 151: IF Component Annotation Only Specification................................. 8
Figure 152: 'IF' Component Software Annotation Behavioural Specification ............. 8
Figure 153: 'IF' Component Deadlock-Free Assertion ................................................. 8
Figure 154: Super Type Component Limits the Behaviour of the Implementation ......... 8
Figure 155: Expected 'IF' Component Behaviour is Deadlock-Free ......................... 8
Figure 156: Expected 'IF' Component Behaviour is a Refinement of Super Type.......... 8
Figure 157: Correctly Driven 'IF' Component Behaves as Expected ......................... 8
Figure 158: Annotating outer level of 'IF' component does not introduce deadlock....... 8
Figure 159: 'IF' Component High Level Behaviour is Deadlock-Free ....................... 8
Figure 160: 'IF' Component Behaves Similarly to Expected Higher Behaviour ............. 8
Figure 161: Higher Software Specification is a Refinement of the Super Type .......... 8
Figure 162: Graphical representation of simplified "IF THEN ELSE" component .......... 8
Figure 163: CSP Model of "IF THEN ELSE" Component ........................................... 8
Figure 164: Refactored correctly driven circuit...................................................... 8
Figure 165: Assertions demonstrating low level specification equivalence................. 8
Figure 166: Hybrid Generic Control Flow Process .................................................... 8
Figure 167: Hybrid Generic Boolean Process ........................................................... 8
Figure 168: Low Level 'IF' Component Specification with Embedded Full Annotations... 8
Figure 169: Hybrid annotations behave identically to annotation only specification....... 8
Figure 170: Check if Outer Annotation Behaviour from Hybrid Model is Allowed......... 8
Figure 171: Hybrid model with annotations renamed behaves identically to correctly
driven circuit................................................................................................................. 8
Figure 172: Overview of proof structure for pattern matched selective renaming alternative ..................................................................................................................... 8

Figure 173: D-Type FlipFlop with no Reset function ......................................................... 8

Figure 174: Generic Control Flow Specification - Correct Driving Limiter ...................... 8

Figure 175: Low Level Generic Control Flow Specification .............................................. 8

Figure 176: Low Level Generic Control Flow Specification with Explicit Deadlocking ... 8

Figure 177: Generic Control Flow Annotation Specification .............................................. 8

Figure 178: Low Level Boolean Specification with Explicit Deadlocking .......................... 8

Figure 179: Generic Control Flow Annotation Specification .............................................. 8

Figure 180: Low Level 'IF' Component Desired Specification ........................................... 8

Figure 181: Low Level 'IF' Component Generic Specification with Explicit Deadlocking ... 8

Figure 182: IF Component Annotation Only Specification ................................................. 8

Figure 183: How to compute the completion time for a 'While' component ....................... 8

Figure 184: Sending and receiving started at the same time on a 0 communication delay channel ...................................................................................................................................... 8

Figure 185: Sending started 1 cycle before receiving on a 0 communication delay channel ...................................................................................................................................... 8

Figure 186: Sending started 2 cycles before receiving on a 0 communication delay channel ...................................................................................................................................... 8

Figure 187: Reading started 1 clock cycle before sending on a 0 communication delay channel ...................................................................................................................................... 8

Figure 188: A simple SCADE model using enumerated types ............................................ 8

Figure 189: Modifying Figure 188 to break the design verifier .......................................... 8

Figure 190: Simplifying the model - Version 1 ................................................................... 8

Figure 191: Simplifying the model - Version 2 ................................................................. 8
List of Tables

Table 1: Model-Checking Performance of FDR ................................................................. 8
Table 2: Port ordering imposed by the Comparator for the tidying algorithms ............. 8
Table 3: Signal ordering imposed by the Comparator for the tidying algorithms .......... 8
Table 4: Definitions of enumerated type used in Figure 188 ........................................ 8
Acknowledgements

I would like to show my appreciation towards my supervisor Dr Roger Peel for his limitless patience and support, without whom I would not have contemplated starting this thesis. Thanks must also be given to Professor Steve Schneider and Dr Helen Treharne for their valuable feedback, support and discussions. Gratitude should also be given to EPSRC, as without their funding none of this would have been possible.

On a more personal note, I would like to thank my parents for their emotional and financial support, during both this PhD and the preceding MSc which lead to this opportunity. Last, but far from least, I would also like to thank my fiancé for her understanding, care and consideration, especially when it came to organising holidays.
1 Introduction

As the number of configurable logic blocks (CLBs) that are contained within field programmable gate array (FPGA) devices continues to increase, this promotes the design and development of larger and more complex systems. A problem then arises that, as the size of systems increase, so does the complexity of the work required for the design and configuration these devices. When FPGAs are traditionally designed, developed and debugged at a low level, through specifying how individual logic blocks (AND gates, OR gates, Flip-Flops, etc) are connected together, the work that is required for traditional verification and debugging involves numerous test vectors and complex simulations that eventually become impractical.

FPGAs are built up from low level hardware components, so they are particularly suited to systems or applications that contain fine grained parallel components. This thesis focuses on methods for creating such fine grained parallel logic circuits from an OCCAM [SGS-Thompson Microelectronics Ltd, 1995] specification. The work reported in this thesis involved the creation of an OCCAM-to-FPGA compiler that builds the logic circuits through performing a one-to-one mapping of OCCAM grammar fragments into small segments of logic circuits, which are then connected together in the manner dictated by the application's specification. The compiler utilises the same code to generate both the logic circuits and formal CSP (Communicating Sequential Processes) [Hoare, 2004] models which describe this generation process and validate the output (see Figure 1).

![Figure 1: How the CSP proof fits with the hardware compilation](image)

The formal CSP models describe the behaviours of small sections of logic circuits that represent segments of the OCCAM language (at various levels of abstraction) and the relationship between these models, along with how these segments are combined together to construct the overall circuit. This enables the compilation of an OCCAM specification...
down into a hardware implementation with a guarantee that the behavioural properties of
the generated circuits will be as dictated by the specification, without the need to
explicitly model check the generated circuit. By proving the compilation process of an
OCCAM-to-FPGA software-to-hardware compiler, one avoids the state space explosion
that would arise if each generated circuit was model-checked. As the compiler has been
designed to utilise the same code to generate both the logic circuits and the CSP models,
we can verify that the compiler output matches the proofs, thus guaranteeing that
properties in the CSP hold true in the hardware circuits.

1.1 Thesis Contributions

This thesis makes a novel contribution to the area of hardware/software co-design through
the development of an OCCAM-to-FPGA compiler that takes a variant of the OCCAM
parallel programming language as its input and produces low level, fine grained parallel
hardware. Importantly, guarantees can be made about the behaviour of the resulting
hardware due to the way the compiler has been developed. By utilising this method of
hardware design, it enabled the production of a formally verified compiler that generates
logic circuits where the behaviour is guaranteed to be as specified. This guarantee of
behavioural properties, without the need to explicitly check each generated circuit,
enables one easily to design and develop larger and more complex systems. The design
and development of these systems are possible because one is working at a higher level of
abstraction (i.e. at the OCCAM software level as opposed to physical hardware). As the
compiler guarantees its output, one can formally verify an application at the software
level as opposed to the hardware level, thus the extra state introduced by the hardware
does not need to be explicitly checked. This is useful because the work involved in
proving the compiler is related to the size and complexity of the software specification
language (i.e. OCCAM), not the application being developed, and this proof only has to
be done once. As a result of this, the end user has the ability to utilise software design and
development methodologies to sub-divide their problem into as many layers of
abstraction or sub-components as desired, without having to worry about manually
verifying the generated hardware.

CSP (Communicating Sequential Processes, [Hoare, 2004]) is a process algebra that
supports modelling the behaviour of interacting processes that execute in parallel. In this
thesis, the components of circuits are modelled as individual processes and then run in
parallel, thus giving a model of the circuit that can be analysed and provides the ability for various properties to be checked. Although modelling a logic circuit in its entirety means that desired and undesired behaviours and specific properties can be explicitly checked [Peel and Cook, 2000], as the size of the application increases, so the state space explosion caused by the fine grained parallel nature of the circuit becomes unavoidable, thus resulting in this model checking approach becoming impractical. Even though this technique becomes infeasible as the size and complexity of the application increases, it does work for small circuits and as such this has formed the basis for the work of proving the compiler's hardware generation process presented in this thesis.

The process of proving the compiler, and thus the automated method for converting software into fine grained parallel hardware circuits, has been achieved by utilising the grammatical structure of the OCCAM language to dictate how small segments of circuit are composed together. CSP models have been used to describe small segments of logic circuits corresponding to each grammatical construct, while proving that each segment of circuit achieves the desired behaviour that the corresponding component requires. These segments of circuit are then proven to be able to be composed together as specified by the OCCAM grammar. The combination of these proofs enables the software specification to be directly converted into a logic circuit through a one-to-one mapping, whilst guaranteeing that the behaviour of the circuit is as specified.

Through guaranteeing that the logic circuits created by the compiler will always conform to the supplied software specification, one avoids having to model-check each logic circuit that is created. This is possible because the compiler has been designed in such a way that the same code generates both the eventual logic circuits and the associated CSP models and proofs. This design choice provides a guarantee that the models and proofs that specify how the circuits were designed directly match how the compiler generates its output. Thus, because the compiler itself generates the models and proofs, this ensures that the proofs and compiler's actions match at all times.

1.2 Possible Utilisation

Designing and developing an application through the use of a software language, rather than custom-designing parts in hardware, simplifies the ability to sub divide the parts of the tasks that will run on a processor or be converted into hardware (software/hardware
co-design). By having the whole application specified at the software level, combined with the ability to convert this software specification into hardware, the component parts may be reclassified from 'running on a processor' to being 'converted into hardware', and vice versa. This re-classification could depend upon the available resources and desired performance requirements. The software components designated to be run on a processor can be implemented in one of two ways – either on a host processor (e.g. a PC) or on a special-purpose built processor on part of an FPGA [Page et al, 1998].

The process of designing hardware at the software level is particularly useful when considering FPGAs that can have sections dynamically reprogrammed or restructured while still running (reconfigurable computing, e.g. [Romer et al, 2000]). Working at a higher level of abstraction than that of the electrical component level would simplify the process of selecting how to split the application into blocks to be re-programmed onto the FPGA chips, along with the conditions under which these reconfigurations should take place. The most obvious direct benefit resulting from this thesis is the simplification of programming FPGAs at a higher level of abstraction, leading to the automatic design and development of the logic circuits and the formal verification of the validity of the generated hardware.

1.3 Thesis Outline

Chapter 1 covers a brief introduction to the work covered in this thesis. It states that the software to hardware compiler that has been developed generates both the logic circuits and CSP proofs describing and proving how those logic circuits are generated. The proofs are also tightly integrated into the compiler, guaranteeing that the behaviours of the compiler and proofs are both consistent and valid.

Chapter 2 provides background information concerning the various technologies used in this thesis. This covers topics such as the OCCAM programming language, the formal process algebra CSP, FDR2 which is a tool to perform refinement and deadlock checks on CSP models, digital logic, FPGAs and methods for programming them.

Chapter 3 is work that directly builds upon [Peel and Wong, 2004]. It automates the previously hand generated process of modelling the overall behaviour of the logic circuits, through modelling the individual hardware components of the circuit and running them in parallel. The result of this work shows that although this technique is feasible for
small logic circuits, the process does not scale. Model checking the generated CSP model of the circuit, for various properties, rapidly suffers from state space explosion as the circuit size increases. This problem arises because of the large amount of fine grained parallelism that is unavoidable when modelling the circuits at the hardware level. Thus a small increase in the amount of hardware a circuit uses will have a dramatic increase in the amount of state space that has to be checked.

Chapter 4 utilises the fact that the work covered in chapter 3 is feasible for small circuits. The compiler was completely redesigned and rebuilt, as part of this research, to take advantage of this. Through utilising the grammatical structure of the OCCAM language to dictate how an application is converted into hardware, the conversion of each grammatical construct can be proven independently of one another. The small segments of circuit for a grammatical construct can be modelled using the technique covered in chapter 3, enabling it to be proven valid and correct. By modelling, and thus proving that each component performs the task that it represents, it is possible to guarantee that combining instances of these grammatical constructs in a one-to-one mapping to that of the application’s OCCAM specification, guarantees that the resultant circuit will behave as required. This is achieved without having to explicitly check the generated circuit, thus avoiding the state space explosion as the circuit size increases, as the amount of state that is required to be checked is dependent on the size and complexity of the OCCAM language, and not the application being converted. This is possible because the technique only requires that each grammatical construct be proven once, and using type checking and inheritance available within Java ensures that components can only be connected together in structures allowed by the OCCAM grammar. As stated previously, the compiler was required to be redesigned and re-implemented to be able to fully utilise this technique, as the previous version did not contain well defined boundaries between grammatical constructs. This redesign and reimplemention of the compiler took advantage of tightly integrating the logic generation with the generation of the models and proofs, which the technique covered in this chapter requires, thus enabling one to guarantee the behaviour of the proofs and the compiler match.

Chapter 5 demonstrates a small example. Commstime is a program, comprising of four parallel processes that circulate data values using channel communication. It presents the OCCAM specification, the corresponding abstract syntax tree (AST) and an analysis of how the grammatical constructs are nested, how they trigger each other (i.e. a
demonstration of when components start and complete). An alternative implementation of
commstime is then presented (along with its trace), that utilises pipelining to slightly
reduce the cycle time of the program. A step through analysis is then presented,
demonstrating how optimising a single grammatical construct would improve the
performance of the application. This chapter also presents a larger example, a digital
clock that accepts user input to alter the time, whilst simultaneously displaying the time on
an LCD display.

Chapter 6 states that, as expected, the compiler worked first time. With the hardware
generation process of the compiler being formally modelled, the output logic circuits are
guaranteed to perform as expected. Thus as the CSP proofs have all been run through
FDR correctly, the only possible outcome was that the compiler would work first time.

Appendix A covers the other supporting work that has been performed in this research,
but is not critical to the comprehension of the main body of the work. This appendix
focuses mainly on the combined digital logic circuit simulator and graphical logic circuit
visualiser that were developed to aid the debugging of the initial compiler. This section
also explains the two custom-developed heuristics that the visualiser can utilise to
simplify the visual layout of the logic circuits within this tool.

Appendix B contains the CSP models of the low level hardware components that were
utilised for the work covered in chapter 3 and in [Peel and Pizarro, 2005]. Appendix C, on
the other hand, contains the low level models representing the hardware components that
were used for the work covered in chapter 4. The reason why an alternative set of CSP
models was created for the work covered in chapter 4 was because, for the redeveloped
compiler to be proven to be correct, the models of the hardware components must also
model the reset functionality to be able to demonstrate that the compiler wired it up
correctly. The work in chapter 3 did not model the reset functionality in an attempt to help
to minimise the state space explosion.

Appendix D presents the CSP models and assertions to prove and define the allowed
behaviour for an example of a single type generic component (these component
categories are discussed in section 4.3 and 4.5). This appendix contains the various
models at numerous levels of abstraction (a hardware behavioural model, a higher
abstract behavioural model, a combined higher abstract and hardware behavioural model),
together with processes to manipulate the models and show consistency between them,
Introduction

along with various assertions to demonstrate and prove desired properties. This component is the super type of the single type component presented in Appendix E.

Appendix E provides the CSP models and proofs for an example of a single type component, which is a sub-type of the example given in Appendix D. This section contains CSP models representing the generated hardware, along with the expected behavioural models of that hardware. This section contains proofs that demonstrate that this implemented grammatical construct is both in itself valid, and also a valid refinement and sub-type of its super-type component contained in Appendix D.

Appendix F and Appendix G are similar to Appendix D and Appendix E respectively, but where the components that they represent are multi-type components. These multi-type components can be variables that have the ability to be both set and/or read from, or channels where data can be transmitted and received. Apart from the CSP to represent the various models and the relationships between them, this section also contains proofs to demonstrate that the individual interfaces of these multi-type components are valid refinements and implementations of their corresponding single type specifications. The models contained within this section demonstrates and proves the properties of how instances of these interfaces interact with each other.

Appendix H and Appendix I are similar to Appendix D and Appendix E, but where the method for obtaining the higher behavioural abstract models has been adapted. Through slightly modifying the method for annotating the hardware behavioural model, the higher behavioural abstraction model can be tailored to give a more software-oriented perspective, as opposed to the hardware-oriented view that is presented in Appendix D and Appendix E.

Appendix J presents a simplified description of the proof framework which would be possible if the reset functionality were not to be considered in the model. The reason why the reset functionality causes the implemented proof framework covered in Appendix D through to Appendix G and chapter 4 to be more complex than the work covered in this section, is because a single higher behavioural abstraction event maps onto the combination of multiple parallel low level hardware events. This causes a problem, as the combination of separate multiple distinct parallel events cannot be mapped directly to a single event. Also, as the hardware generation process of the compiler is required to
connect the reset signals in the logic circuits it generates, to prove the generation process the models used in the proofs, it must be able to contain and represent this behaviour.

Appendix K contains some supporting CSP models for the example that the speculative work in Appendix J requires.

Appendix L defines the various clock cycle implementation times for the implemented components and any dependencies upon any internal components that they require. It also demonstrates graphically the timing for the on-chip channel communication that has been implemented.
2 Literature Review

In this chapter, the literature that provided the motivation for the research is identified, and an overview of the different languages used in the thesis: OCCAM and CSP together with their supporting tools. Other approaches to hardware/software co-design are also compared, which start with a description of the proposed system using a programming language.

The research in this thesis is based upon [Peel and Cook, 2000], initially extending their approach by automating the CSP model creation of the logic circuits (see chapter 3 and [Peel and Pizarro, 2005]) which were previously hand crafted. These papers chose to utilise OCCAM (see section 2.4) as the starting language to specify an application that is converted into parallel hardware for programming onto an FPGA (see section 2.1) and CSP (see section 2.2) as the formal algebra that is used to create the models and perform the proofs. This is a natural combination, as OCCAM [SGS-Thompson Microelectronics Ltd, 1995] was initially inspired by CSP [Hoare, 2004].

Similar work, covering the conversion of OCCAM or OCCAM-like languages into hardware for programming onto FPGAs, has been attempted previously. Ian Page and Wayne Luk built a prototype OCCAM to FPGA compiler in Prolog [Page and Luk, 1991], with the choice of using Prolog being that “the implementation and proof are very close to each other”. The utilisation of the phrase “very close to each other” combined with the lack of any mention regarding integrating the proof directly into the compiler does not guarantee that the implementation matches the proof, even though the theory of how the compiler works may be proven. The framework presented in [Schenke and Dossis, 1997], on the other hand, is a denotational semantics of the Handel-C language, stating that “It is the ambition of the authors work to supply a transformational framework in which an implementation relation can be defined between Handel and hardware ...” (Handel-C is defined in [Agility Design Solutions Inc. 2007] and [Ian Page, 1997]). This framework demonstrates the theory that the compiler is built on, but it has not been integrated into the compiler. Thus, it does not guarantee that the compiler matches the proofs. If the compiler conforms to the rules, it will generate the hardware correctly but no guarantee is given that this is the case. For example, whilst the code for
literature review

creating the hardware may contain a subtle bug, the current maturity of the tool helps to build confidence that this is not the case, but it does not guarantee it.

Handel-C was initially based on OCCAM; its compiler was commercialised in early 1996 as the main product for "Embedded Solutions Limited" (which in 2000 was renamed to Celoxica, who recently have sold the DK Suite that contains the Handel-C compiler to Agility Design Solutions Inc). As the Handel-C hardware compilation tool has never been promoted as being formally proven, one can assume that the proofs covered in [Schenke and Dossis, 1997] were never integrated into the compiler. This conclusion is based on the fact that tightly integrating the proof into the compiler impacts upon the design. The paper states that "Handel is a language already used by hardware designers" and that this technique has the "potential" for integration into hardware compilation systems, the implication that it was not integrated into the compiler at the time of writing the paper can be deduced. Integration of the technique into the compiler would have resulted in a considerable refactoring of the compiler code, work that is both highly detailed and time consuming. To the best of our knowledge, the Handel-C compiler remains unproven.

2.1 FPGA

A field-programmable gate array (FPGA) is a type of programmable logic chip. The device is comprised of an array of configurable logic blocks (CLBs) that can be connected together by a programmable connection framework, thus representing a clocked digital circuit. Apart from being able to represent thousands of simple logic blocks ('and' gates, 'or' gates, 'flip-flops' etc) interconnected together, FPGAs can also contain devices such as special data processing units, memory blocks and complete processors such as the PowerPC [Xilinx, 2005].

2.1.1 Programming FPGAs

There are numerous ways to program FPGAs [Wain et al, 2006], with manufacturers usually providing a basic schematic tool that takes a low level circuit, performs optimisations and converts the result into a format for 'place and routing' (i.e. programming or configuring) of the FPGA chips. These tools, although they tend to provide a graphical representation for the development and specification of a circuit, usually generate low level circuit specifications or netlists e.g., Electronic Design
Interchange Format (EDIF) [EDIF Steering Committee, 1988] of the circuit to program onto the FPGA. The problem with the task of designing, developing and reasoning about circuits at this low level is that the required work tends to scale poorly and becomes much more difficult as the complexity of the system being developed increases. As mentioned previously, the use of simulations and test vectors [Riesgo et al, 1998] becomes impractical as the circuit size increases.

The VHSIC Hardware Description Language (VHDL, [IEEE Std-1076, 2002]) and Verilog [IEEE Std-1364.1, 2002] design languages were developed to simplify the design and specification of digital hardware. They provide a way to specify a hierarchy of modules through specifying the physical and/or behavioural characteristics of the logic being built. JHDL [Bellows and Hutchings, 1998], like VHDL, is another hardware description language. Each type of logic element is represented by a Java class, and Java methods are used to specify how instances of these classes can be interconnected together. Typically, the method utilised for testing HDL specified components and evaluating their interactions is through simulation, checking every permutation of the signals that can occur. This is a very demanding and computationally expensive process. Even though VHDL is currently one of the main languages used to design FPGAs, as the size and complexity of the systems one wishes to create increases, new development methods and languages are going to be needed to help one cope with designing and developing them.

SystemC [IEEE Std-1666, 2005] is a C++ class library that has been built to have hardware orientated constructs. It utilises a thread-like mechanism for parallelism, with event methods to provide synchronised communications Alternatively, Handel-C ([Agility Design Solutions Inc. 2007] and [Ian Page, 1997]) is a programming language with C-like features, but is similar to CSP in the fact that when two processes need to exchange data they both have to perform the relevant I/O task synchronously. If both processes are not ready to exchange that data, then one waits for the other. When both processes have completed the data transfer, they can continue to perform their own tasks.

2.2 CSP

CSP (Communicating Sequential Processes) [Hoare, 2004] is a process algebra. Its main use is in the description of parallel processes and their interactions. CSP, and its machine
readable form CSP\textsubscript{M}, can be manipulated to check events or prove various properties of the system being modelled. To assist with performing this task, several automated tools are available (e.g. FDR [Formal Systems Ltd, 2005], see section 2.3). The investigation of various properties of parallel processes is achieved through examining all permutations of how these parallel processes are serialised, with each specific ordering of the execution of the events being a specific trace. It is through examining these traces that one can determine if particular properties hold true or not, a fact that has been used within this thesis to examine models that represent logic circuits and the software to hardware conversion process.

2.2.1 Processes and Events

A process can be comprised of zero or many events, followed by another process. The events represent actions or communications, specified as atomic names (e.g. on or off), compound names (e.g. light.on or light.off), or channel input/output events (e.g. lightswitch?x or sendstate!x). Various algebraic operators can be used to build up and specify more complex behaviours, and some of these are described below.

E.g. A process \( P \) that can do event \( a \) followed by event \( b \), an infinite number of times, is defined as:

\[
P = a \Rightarrow b \Rightarrow P
\]

2.2.2 External Choice: Deterministic Choice

External choice is a branching point, enabling a process to offer multiple choices of events it is prepared to perform. The choice of which path gets executed is determined externally from the process.

E.g. A process \( R \) that is defined as event \( a \) or \( b \) followed by process \( P \), or event \( c \) followed by process \( Q \), is defined as:

\[
R = a \Rightarrow P
\]
\[
\begin{align*}
& a \Rightarrow P \\
& b \Rightarrow P \\
& c \Rightarrow Q
\end{align*}
\]
2.2.3 Internal Choice: Non-deterministic Choice

Internal choice is a branching point, where the process chooses which of the multiple options to offer. This differs from external choice, where the process offers all the available options.

E.g. A process $R$ that is defined as a process that offers event $a$ followed by process $P$, or a process that offers event $b$ followed by process $Q$, is defined as:

$$R = a \rightarrow P$$
$$b \rightarrow Q$$

It should be noted that process $R$ behaves the same as:

$$R = \tau \rightarrow a \rightarrow P$$
$$\tau \rightarrow b \rightarrow Q$$

Where $\tau$ (tau) is a hidden event.

2.2.4 Hiding

The hiding operator provides a mechanism to enable specific events to be hidden (i.e. to be unobservable).

E.g. A process $R$ that is defined as event $a$ followed by process $P$, with $a$ events hidden, is defined as:

$$R = (a \rightarrow P) \setminus \{a\}$$

If process $P$ does not contain any $a$ events, then process $R$ behaves as process $P$. If process $P$ does contain $a$ events, then this is not the case, because the $a$ events within this instance of process $P$ are also hidden. E.g. If $P$ is defined as:

$$P = a \rightarrow b \rightarrow P$$

Then process $R$ would behave as if it was defined as:

$$R = b \rightarrow R$$

2.2.5 Boolean Guards

Boolean guards provide a mechanism to only offer options under specified conditions.

E.g. A process $R$ that is defined as the process $P$, guarded by the condition $g$, is defined as:

$$R = g \wedge P$$

This process will only offer to perform the process $P$, if the condition $g$ evaluates to true.
2.2.6 Let Within

‘Let Within’ blocks enable the scoped definition of sets and processes, but not channels.

E.g. A process \( R \) locally defines a process \( S \), which can only be used within process \( R \), is defined as:

\[
R =
\begin{array}{l}
\text{let} \\
S = a -> S \\
\text{within } S
\end{array}
\]

The process \( S \) cannot be used outside process \( R \), so the following defined process is illegal, as process \( S \) has not been defined:

\[
Q = S
\]

2.2.7 Alphabetised Parallel

Running two or more processes in alphabetised parallel, each process can only perform the events specified for them, and they are also required to synchronise on the common events in the alphabets.

E.g. A process \( R \) that is defined as process \( P \) and \( Q \) run in parallel with their corresponding alphabets, is defined as:

\[
R = P \{ a, b \} || \{ b, c \} \} Q
\]

This defines that process \( P \) can perform events \( a \) and \( b \), process \( Q \) can perform the events \( b \) and \( c \), with the processes requiring synchronisation on \( b \) events. If process \( P \) and \( Q \) are defined as:

\[
P = a -> b -> P \\
Q = c -> c -> b -> Q
\]

Process \( R \) behaves the same as if it was explicitly defined as:

\[
R = a -> c -> c -> b -> R \\
\text{[]} \\
c -> \{ a -> c -> R \} \\
\text{[]} \\
c -> a -> R \\)

2.2.8 Shared Parallel

Running two or more processes in shared parallel requires that the processes synchronise on the specified events. All the processes can only perform the specified events, if all the processes perform the event at the same time.

E.g. A process \( R \) that is defined as process \( P \) and \( Q \) run in parallel synchronising on event \( b \), is defined as:

\[
R = P \{ b \} || Q
\]
If process P and Q are defined as:
\[ P = a \rightarrow b \rightarrow P \quad Q = c \rightarrow c \rightarrow b \rightarrow Q \]
Process R behaves the same as if it was explicitly defined as:
\[ R = a \rightarrow c \rightarrow c \rightarrow b \rightarrow R \]
\[ \; c \rightarrow (a \rightarrow c \rightarrow R) \]
\[ \; c \rightarrow a \rightarrow R \]

2.2.9 Interleaving

Interleaving two or more processes enables the processes to execute their events independently of each other. A trace, the sequence of events that have occurred, can be built up from executing from any of the processes any of the possible events that they can next perform.

E.g. A process R that is defined as processes P and Q interleaved, is defined as:
\[ R = P || | Q \]
If process P and Q are defined as:
\[ P = a \rightarrow b \rightarrow \text{SKIP} \quad Q = c \rightarrow b \rightarrow \text{SKIP} \]
The possible traces for process R are:
\[ \langle a \rangle \]
\[ \langle a, b \rangle \]
\[ \langle a, c \rangle \]
\[ \langle c, a \rangle \]
\[ \langle c, b \rangle \]
\[ \langle a, b, c \rangle \]
\[ \langle a, c, b \rangle \]
\[ \langle c, a, b \rangle \]
\[ \langle c, b, a \rangle \]
\[ \langle a, b, c, b \rangle \]
\[ \langle a, c, b, b \rangle \]
\[ \langle c, a, b, b \rangle \]
\[ \langle c, b, a, b \rangle \]

2.2.10 Congruence & Monotonicity

The semantics are congruent if for all operators within the language (i.e. CSP), the result can be computed from the the application of it on its component parts. i.e. "it is possible to compute \( S[[P \oplus Q]] \) in terms of \( S[[P]] \) and \( S[[Q]] \)" [A. W. Roscoe, 1998].

For the context of a process to be monotonic, if process P is contained within process Q, then the context of P must also be contained in the context of Q. i.e. "\( P \subseteq Q \Rightarrow C[P] \subseteq C[Q] \)."
2.3 FDR

FDR (Failures-Divergence Refinement) is a CSP refinement checking tool [Formal Systems Ltd, 2005] produced to check and establish properties of CSP models. Through testing for refinements of the system (see below), it can check to see if specific properties hold true or not. It is able to systematically check all the possible states and transitions that the system can reach and perform, being able to determine if the system is deterministic, deadlock free (i.e. the CSP model can never reach a state where there are no events to perform) and/or a valid trace refinement of another CSP model. This thesis uses FDR to examine, analyse and compare against each other, the CSP models used to represent the logic circuits and software to hardware compilation process.

A CSP process is a valid trace refinement of another process (‘P \[\triangleright_Q\] or ‘Q is a trace refinement of P’), if all the allowed orderings of events that can occur also exist in the process it is being trace refined against (i.e. all traces that Q can produce are contained within the traces that P can produce). A process is a failures refinement of another (‘P \[\triangleright_F=Q\] or ‘Q is a failures refinement of P’), if for all points within its traces, all sets of events that it can refuse at that point are contained within the sets of events that are refused at the corresponding point within the trace of the process it is a refinement of (i.e. for all points within the traces that Q can perform, the sets of events that is can refuse to do are contained within the sets of events that P can refuse when at the same point within its traces). For a process to be a failures-divergences refinement of another (‘P \[\triangleright_{FD}=Q\] or ‘Q is a failures-divergence refinement of P’), apart from the set of traces and refusals having to be contained within that of the process it is refining, the set of traces that dictate when it can get into a livelock state (i.e. performing an infinite sequence of internal events) is contained within the set of traces that the process it is refining can get into livelock.

2.4 OCCAM

Occam is what is commonly referred to as a very fine grained parallel programming language. It was initially developed by a team at INMOS in conjunction with the design of the transputer processor [INMOS 72-TRN-048-03, 1987], and it is based on C. A. R. Hoare's CSP (see section 2.2) [Hoare, 2004].
Occam supports programming at a very fine grained level both the parallel and sequential nature of a program. Communications between the parallel processes is achieved through passing messages along channels. The static memory model with Concurrent Read, Exclusive Write (CREW) parallel usage [Quinn, 1994] enables the software to be verified free from parallel usage errors, which are errors that can occur when channels or variables are written to or read-and-written to in parallel.

Apart from Occam being used as the initial basis of Handel-C, the principles of Occam have also been implemented numerous times in Java libraries (e.g. JCSP [Austin, 1998] and CTJ “Communicating Threads for Java” [Hilderink, Bakkers and Broenink, 2000]. CTJ was renamed from CJT “Communicating Java Threads” for legal reasons, thus it may also be referred to in older work by it previous name). It is clear from this that the conceptual principles and properties obtainable from the Occam and CSP programming style are still considered a worthwhile avenue for practical exploration and utilisation.

2.5 Clocked Logic

Clocked logic is the particular type of logic currently produced by the compiler detailed in this thesis. This particular type of logic utilises a global clock to control the progression of signals through the circuit by controlling when flip-flops output their next signal. With the possibility of non-clocked combinatorial logic (e.g. ‘and’, ‘or’ and ‘not’ gates) also being contained within the circuit, the maximum allowable clocking speed is dependent on the time it takes for signals to propagate through these parts of the system. Although it is permissible to have cycles of logic, forming loops, these loops should not normally be solely made up of non-clocked logic in a fully-clocked design. If circuits contain non-clocked cycles, then race conditions may occur within the circuit, a fact that is often undesirable but can be used quite safely in components such as transparent latches.

2.6 The IO-PAR Design Paradigm

[Peel and Wong, 2004] determined that the physical behaviour of the components from which clocked logic circuits are built behave in an IO-SEQ and IO-PAR manner (combinatorial logic and clocked logic, respectively), with the combined behaviour being that of the IO-PAR design paradigm covered by [Welch, Justo and Willcock, 1993].
Through combining IO-PAR and IO-SEQ components, various properties of the system can be examined, such as deadlock freedom, thus guaranteeing no combinatorial loops.

Both IO-SEQ and IO-PAR processes run forever, inputting and outputting values on a number of channels and performing computation on these values. An IO-SEQ component receives its inputs in parallel before performing its outputs in parallel, whereas an IO-PAR component sends and receives all its inputs and its outputs in parallel. With the IO-PAR components inputting and outputting in parallel, any combination composed together as a network is guaranteed to be deadlock-free. From an external point of view, the network will be indistinguishable from a single IO-PAR component.

An IO-Rnet is a network containing IO-PAR and IO-SEQ components, with an IO-SPnet being a special type of IO-Rnet. For a network to be an IO-SPnet it must contain no loops consisting of only IO-SEQ components, and no paths from an external input to an external output of only IO-SEQ components. Therefore externally the IO-SPnet is indistinguishable from an IO-PAR component, and as such it is guaranteed to be deadlock-free.

2.7 SableCC

As stated earlier in Section 2, this thesis is motivated by Peel's original OCCAM compiler which uses the SableCC [Gagnon and Hendren, 1998] parser generator. SableCC is a parser generator with an object-oriented structure, written in Java. By supplying it with a LALR1 grammar (the OCCAM grammar required refactoring and the addition of several annotations to comply with SableCC's requirements), SableCC can automatically generate the parser and AST (abstract syntax tree) for a compiler. Through the use of object-oriented techniques, the output that it produces is in a well-structured format, with the class inheritance and naming conventions of the produced Java classes being directly related to the production rules and annotations within the grammar. This linkage of the output to the grammar means that both minor modifications and expansion of the grammar can be achieved with minimal fuss and the changes in the output from SableCC can be predicted from changes within the grammar.

The predictability of changes within SableCC's output means that it is particularly suited for use with prototyping. This means that software written to utilise sections of the generated parser that have not changed can continue to be used when modifications to the
grammar have been made. As the parser, abstract syntax tree (AST), and custom executable code are separate from each other, the work presented in this thesis is relatively independent of the parser generator being used.

2.8 Comparison with Related Work

This section highlights other related research that covers different aspects of the work presented in this thesis. Firstly, the work covered in this section describes underlying languages with formal semantics. This enables the analysis of the behaviour of the software specification, which contrasts with the approach of my thesis, which concerns itself with the production of verified hardware and the integrated method of validating it. In this thesis, the models covered are used in the analysis of how the logic is created, providing a proof of construction. Secondly, this section examines various tools that contain verification functionality through model checking.

The work covered in [Butterfield & Woodcock, 2006], provides denotation semantics of a "possible (very naive) hardware implementation" of Handel-C. This is achieved through modelling the hardware as a finite state machine, whereby it is defined as a single fixed function describing how the state changes. With the technique being created independently from a practical tool (e.g. the Handel-C compiler), the technique enables one to analyse how hardware may be created, but it does not guarantee that a tool builds hardware in this manner and therefore can be considered foundation work. Whereby the more recent work covered in [Perna & Woodcock, 2008] describes the "limited progress" that has been made towards both an axiomatic semantic definition of Handel-C and algebraic rules to describe the characteristics of a Handel-C program, with their work tending to the unification of the various semantic models of Handel-C.

Augmenting existing tools with formal verification capabilities is common in several industrial tools. For example, [Hamon, 2008] describes the Simulink Design Verifier, a verification tool for Simulink enabling the user to automatically create test cases and check for the existence of various behavioural properties. Similarly [Köhler and Kant, 2004], describes a verification tool for Esterel SCADE. Both verification tools have been implemented using the Prover Plug-In [Sheeran, 2000], a product developed by Prover Technology (http://prover.com) based on Stålmarck's method of tautology checking using propositional logic [Harrison, 1996] and [Sheeran & Stålmarck, 2000].
Augmenting a tool after it has initially been designed and developed with the ability to formally model properties of a system, would require that the tool perform a translation of a defined system from one representation to another. This means that an instantiation of the internal data structure of the tool (uses to represent a system), would have to be converted to the data structure that the proof tool requires so that properties of the model could be checked. This unseen conversion from one data structure to another has the potential to invalidate the formal verification if errors or bugs were to occur. In essence, these errors can have two main effects, which are:

1. That behavioural properties (desired and/or undesired) which are specified by a system, may not be represented within the model that is formally checked (i.e. the formally checked model may not be the model that was specified in the tool, or the resultant output of the tool).

2. That a defined system can cause the translation process to fail, resulting in not obtaining a formal model to check. This is different from creating a correctly specified model that cannot be analysed due to known limitations of the formal verification process (e.g. due to state-space explosion).

It is the previously experienced difficulties caused by the second point specified above that highlights the existence of the hidden conversion step and the presence of problems with it. For example, the Design Verifier that is part of Esterel SCADE 6.0, which enables the user to check for properties within a system that they have defined using the tool (graphically and/or textually), can be made to fail to create a model for formal verification (although the tool will both simulate it correctly and create the valid resultant output), through defining a model that combines trivially together a collection of simple constructs that the design verifier independently manages to process and analyse (see Appendix M for a simple example).

The removal of the potential for error, within the integration of formal verification into a tool, was part of the aim for the work tackled in this thesis, and it was achieved through designing and developing the tool whilst taking into consideration the desire for tight integration of the creation of the proof at the same time. The code within the software-to-hardware compiler can directly build both the hardware circuit representation and the formal model of that hardware to use in the proof, thus guaranteeing that they match, as the data structure does not have to be manipulated or restructured. Even though this is a
sound and viable approach to take to this problem, it is uncommon as it requires a holistic problem solving strategy, simultaneously taking into consideration all factors relating to the problem.
3 Preliminary Work: Automatic CSP
Modelling of Generated Logic Circuits

The original concept of this research was to automatically model the output of a software-to-hardware compiler which produces fine-grained parallel logic circuits. Due to the results of the preliminary work obtained within this chapter, the research concept was refined in chapter 4. Prior work had demonstrated that a formal model of a logic circuit may be built, from which its behavioural properties can be examined and verified to be correct, thus guaranteeing that the circuit performs the desired task.

The work reported in this chapter covers how the low level logic components are modelled, along with how these models are utilised to produce a representation of the hardware. This chapter also demonstrates how this representation of the hardware can be utilised to validate the generated logic circuits. It also describes some of the limitations that this technique suffers from, along with the early strategies utilised in an attempt to manage these limitations.

Although this technique cannot sensibly scale up as the circuits become larger (due to the modelling technique suffering from state-space explosion), the work explained in this chapter is critical to that of the main body of the research covered in chapter 4.

3.1 Modelling Logic Circuits

As indicated in [Peel & Wong, 2004], a CSP model of a logic circuit can be created. This can be achieved by running in parallel models that represent each of the logic components in the circuit. The circuits' behaviour is explored by simulating the interactions of the hardware components by modelling the circuits' net-list data structure. This provides a formal model of the circuit, which is then examined to prove the existence (or lack) of various properties.
3.2 Modelling Logic Components

The process of modelling the logic components (AND gates, OR gates, FLIP-FLOP's etc) involves building individual processes to represent each component. Because clocked logic circuits are being modelled, the components used fall within two distinct subcategories, clocked and non-clocked logic. The strategy chosen to underpin the modelling of these logic components is the IO-SEQ / IO-PAR model [Welch, Justo and Willcock, 1993], with the non-clocked components being IO-SEQ and the clocked ones IO-PAR.

3.3 Why Use CSP?

One of the initial reasons for utilising CSP to model the logic circuits, is because it was used for the modelling of logic circuits by Peel & Wong [Peel and Wong, 2004]. Their work involved hand crafting CSP models of the individual logic components and running them in parallel, thus enabling them to specify logic circuits. The CSP model of a logic circuit was then compared against a high-level CSP specification of the program's algorithm. Using FDR2 [Formal Systems Ltd, 2005] various properties of the two circuit models could be tested, enabling the circuits' validity and correctness to be verified. More compelling reasons for the utilisation of CSP are covered later, in section 4.4.1.
3.4 Component Models in CSP

The CSP models used in this section of the research takes its inspiration from the models used in the work covered by [Peel & Wong, 2004]. Figure 3 illustrates an IO-PAR behavioural model of a D-Type Flip-Flop, where its inputs and outputs can occur in either order during each clock cycle (i.e. between clock events).

```
D_TYPE(clock, d_in, q_out) =
let
  A(x) =
    q_out ! x -> d_in ? z -> clock -> A(z)
        []
    d_in ? z -> q_out ! x -> clock -> A(z)
within A(0)
```

Figure 3: Simplified D-Type Flip-Flop Model

Appendix B contains the models of the low level logic components used within chapter 3, although the work reported in chapter 4 uses an alternative implementation. The alternative implementations are illustrated in Appendix C, with the reasoning behind the adaptation of the models covered in section 4.4.2.

3.5 Circuit Models in CSP

For each logic component contained within the circuit, a CSP process was created as an instance of the model that describes the behaviour of that component (e.g. Figure 4 illustrates both graphically and in CSP, an instance of an “AND” gate).

```
<table>
<thead>
<tr>
<th>CSP Specification of an AND Gate</th>
<th>A Graphical Depiction</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- Declaration of signal channels that the gate is wired to.</td>
<td></td>
</tr>
<tr>
<td>STATE = {0, 1}</td>
<td></td>
</tr>
<tr>
<td>channel chan_input1 : STATE</td>
<td></td>
</tr>
<tr>
<td>channel chan_input2 : STATE</td>
<td></td>
</tr>
<tr>
<td>channel chan_output : STATE</td>
<td></td>
</tr>
<tr>
<td>-- Declaration of an instantiation of a two input AND gate.</td>
<td></td>
</tr>
<tr>
<td>INST_AND = AND_GATE(chan_output, &lt;chan_input1, chan_input2&gt;)</td>
<td></td>
</tr>
<tr>
<td>-- Alphabet of instantiated AND gate</td>
<td></td>
</tr>
<tr>
<td>ALPHA_INST_AND = [{chan_output, chan_input1, chan_input2}]</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4: An Instantiation of a Two Input AND Gate

These CSP processes were then run in parallel so that they synchronise on events which they share with each other. This synchronisation was achieved by going through all the
processes and running them in parallel, synchronising on any events that the current
process shares with that of any of the preceding processes specified in the
"SYSTEM_LIST". Figure 5 illustrates the CSP required to create instances of the logic
gates and connect them together for the segment of circuit presented in Figure 2 on page
8.

An alternative method for specifying the logic circuits is through writing an algorithmic
specification of the circuit. This algorithmic specification describes the relationship and
dependencies between the circuits' inputs and outputs. An example of this is illustrated in
Figure 6 for the circuit presented in Figure 2 and again in Figure 5.

```plaintext
-- Channel Declarations
STATE = {0, 1}
channel chan_clock
channel chan_1, chan_2, chan_3, chan_4, chan_5 : STATE

-- Logic Components
ALPHA_INST_1 = {I chan_clock, chan_3, chan_1}
INST_1 = D_TYPE(chan_clock, chan_3, chan_1)
ALPHA_INST_2 = {I chan_2}
INST_2 = VCC(chan_2)
ALPHA_INST_3 = {I chan_2, chan_3, chan_4}
INST_3 = AND_GATE(chan_4, <chan_2, chan_3>)
ALPHA_INST_4 = {I chan_clock, chan_3, chan_1}
INST_4 = T_TYPE(chan_clock, chan_4, chan_5)

-- The Circuit Model
SYSTEM_CIRCUIT = CIRCUIT_PAR(SYSTEM_LIST, SYSTEM_ALPHA)
-- List of Components Used & their Alphabets
SYSTEM_LIST = < (INST_1, ALPHA_INST_1), (INST_2, ALPHA_INST_2),
(INST_3, ALPHA_INST_3), (INST_4, ALPHA_INST_4) >

-- Channels to/from Outside Circuit
SYSTEM_ALPHA = {I chan_clock, chan_1, chan_5}

-- Process Used to Run Components in Parallel
CIRCUIT_PAR(component_list, outer_alpha) =
let
  -- Deadlock if no circuit components
  A(<> ) = STOP
  -- If one component, run it hiding inner channels
  A(<(p1, a1)> ) = ( p1 \ diff(a1, outer_alpha) )
  -- If multiple component, run in parallel
  A(<(p1,a1)><(p2,a2)>p3) = A(<(B(p1,a1,p2,a2), union(a1,a2))>p3)
  -- Run two components in parallel
  B(p1,a1,p2,a2) = (p1 [] inter(a1, a2) |) p2
within A(component_list)

Figure 5: Example CSP of Running Logic Components in Parallel
```
Preliminary Work: Automatic CSP Modelling of Generated Logic Circuits

-- Channel Declarations
STATE = \{0, 1\}
channel chan_clock
channel chan_1 : STATE
channel chan_5 : STATE

-- The Algorithmic Specification of the Logic Circuit
SYSTEM_SPECIFICATION =
let
  -- Perform the IO in parallel
  A(x, y) =
    chan_1 ? z -> chan_5 ! y -> B(z, y, x)
  []
    chan_5 ! y -> chan_1 ? z -> B(z, y, x)

  -- Compute the next cycles output
  B(x, y, z) =
    chan_clock ->
      ( z == 1 & A(x, 1-y) )
    []
    ( z == 0 & A(x, y) )
within A(0, 0)

Figure 6: Algorithmic Specification of the Logic Circuit from Figure 5

3.6 Proving the Logic Circuit is Equivalent to a Low Level Specification

By creating a direct representation of the generated circuit (e.g. Figure 5) along with an algorithmic specification of the desired behaviour (e.g. Figure 6), one can test and prove various properties of the hardware using FDR2 (see Figure 7).

-- Check that the modal of the specification is deadlock free
assert SYSTEM_SPECIFICATION : (deadlock free \[F\])

-- Check that the model representing the circuit is deadlock free
assert SYSTEM_CIRCUIT : (deadlock free \[F\])

-- Check that the circuit and specification have the same behaviour
assert SYSTEM_SPECIFICATION [FD= SYSTEM_CIRCUIT
assert SYSTEM_CIRCUIT [FD= SYSTEM_SPECIFICATION

-- If both the failure divergence refinement assertions hold true,
-- this guarantees that the the two previous deadlock free assertions
-- will return the same results.

Figure 7: Simple Testing of the Generated Circuit

The basic testing involves ensuring that the model of the logic circuit generated is equivalent to that of the specification containing the desired and expected behaviour. This is achieved in FDR2 through the use of failures divergent refinement models. The deadlock-free check done in FDR2 helps to ensure that the models do not contain any race conditions or other undesirable properties, this is because the models were designed and constructed so that occurrences of specific properties would cause them to deadlock.
3.6.1 Performance Problems: State Space Explosion

Even relatively simple OCCAM programs compiled into hardware produce large numbers of flip-flops and logic gates. Since simulating these generated circuits requires us to model the system at the hardware level, a straight mapping into CSP quite rapidly leads to an uncontrollable state space explosion when specifications such as those in Figure 5 are model checked. The main work reported in chapter 4 focuses on attempting to bypass this state space explosion. The preliminary work in this chapter covers several simple techniques in an attempt to manage or delay this problem:

- Simplification of the logic component models (see section 3.6.3) helps to alleviate the state space problem, although further techniques can also be used.
- The high quantity of fine grain parallelism, combined with the fact that we are generating clocked logic (where we assume that the choice of clocking speed is such to enable the circuit to stabilise between clock cycles) means that we can also utilise 'CHASE' compression and is discussed in section 3.6.2.

3.6.2 Utilising CHASE to Achieve Viable Runtime Simulations

To enable the simulation of the logic circuits to run within a viable time period, we started by applying CHASE compression in FDR to the outer level of the CSP process that represents each circuit. We can utilise CHASE, which selects one trace order for interleaved tau events, due to how we model and combine the logic components to represent the circuit in CSP.

When instances of the CSP representing each logic component in the circuit are joined in parallel (see section 3.1), the fine grain parallelism contained in the hardware can cause the CSP to have a vast numbers of interleaved events. The reason why we can avoid checking every possible interleaved trace ordering is because CHASE works on the hidden events that represent internal signals between logic components. The way we chose to model the logic components (see section 3.2 and 4.4) causes each channel to trigger once and only once in each clock cycle. If this were not the case, the model would deadlock regardless of the interleaving order of the hidden events. Choosing any one triggering order is sufficient for the CSP to operate, and CHASE achieves this.
3.6.3 Implementation & Model Simplification

Although modelling each of the logic components can be perceived as an individual task, the fact that instances of these models are combined together to represent circuits has implications that affect both their design and their implementation. Since the CSP models of these components are exclusively used to create representations of the circuits that the compiler generates, it is possible to guarantee several of their properties (see below) and to utilise these to simplify the models, thus reducing the state space that is model checked.

- Clocked logic components only need to know the last states on their inputs before the clock event (the low-to-high clock transition). Thus the ability to receive multiple different input values in any one clock cycle can be removed from the model as it is not needed, thereby reducing the state space of the models.

- The clocked components we utilise only alter their outputs on a low-to-high clock transition. Therefore the clock low-to-high transition can be modelled as a single event, as opposed to modelling both the high and low states.

- Each circuit utilises a single global reset that can only be triggered from outside the circuit. Triggering the reset will cause the circuit to revert to its initial state. Thus, the reset signal can be removed as its assertion simply starts the model again. This is because no part of a circuit can carry state over the reset occurrence, and therefore cannot alter the behaviour of its next instantiation. Also, the circuits that the compiler currently generates cannot be self resetting (i.e. they can not trigger their own reset).

The models that represent the logic components have undergone several iterations for two main reasons (see Figure 8 and Figure 3), the first being that I was learning CSPM while I was developing them, and the second was an attempt to reduce the state space that they create. The model covered in Figure 8 was the initial version that I developed, but the properties stated above were identified and the specified simplifications were applied, thus resulting in the model covered in Figure 3 (which results in a model closely resembling that contained in [Peel & Wong, 2004]). Currently the logic component models being utilised in this section of the work are specified in Appendix B, whereas the main body of the work covered in section 4 utilises the models specified in Appendix C. One of the main differences between the different models is that the reset functionality was reintroduced back into the models for the main work covered in section 4, this was so
the models were a more direct representation of the logic circuits and the generation process, so that if the compiler had an error that connected the reset signals incorrectly (which it doesn’t), it would be modelled in the CSP and identified.

```plaintext
channel internal_state : {0, 1}
T_TYPE(clock, reset, d_in, q_out) =
   let
      -- Run the inputs and outputs in parallel (ZO-PAR)
      A = ( ( B [ ] [ ] internal_state, clock [ ] [ ] C ) \ [ ] internal_state [ ] )
      -- Perform the outputs
      B = internal_state ? z -> q_out ! z -> clock ? _ -> B
      -- Perform the inputs & dictate the next clock cycles output value
      C =
         let
            -- In Clock Low State
            CA(x) =
                Let
                    CAA = internal_state ! x ->
                        ( reset ? y -> d_in ? z -> CAB(y, z)
                          [ ] d_in ? z -> reset ? y -> CAB(y, z)
                        )
                    CAB(y, z) =
                        clock ? 0 -> CAA
                        [ ]
                        clock ? 1 ->
                            y == 1 & CB(0)
                        [ ] y == 0 & CB(z)
                    within CAA
            -- In Clock High State
            CB(x) = internal_state ! x ->
                ( [ ] [ ] y: {reset, d_in} @ y ? _ -> SKIP )
            ;
                ( clock ? 0 -> CA(x)
                  [ ] clock ? 1 -> CB(x)
                )
            within CA(0)
        within A
Figure 8: Initial T-Type Flip-Flop Model Developed
```

3.7 Experimental Results

An example of the automated CSP generation and circuit proof can be demonstrated by comparing three different implementations of the same task, together with a hand crafted CSP specification. The chosen software task is a cyclical counter that outputs a stream of consecutive integers along a channel. Figure 9 specifies the outputs directly.

```plaintext
SPEC =
   out15 -> out16 -> out17 -> out18 -> out19 -> out10 -> out11 -> out12 ->
   out13 -> out14 -> out15 -> out10 -> out11 -> out12 -> out13 -> out14 ->
SPEC

Figure 9 CSP specification of a 4-bit counter process
```
The various implementations each output the same order of values but at slightly different rates. Even though the implementations achieve the same task, the difference in performance is caused by their differing structure and amount of parallelism which is dependent on the OCCAM code that was used to compile the hardware along with the efficiency of the generated logic.

The next implementation (CTR) is generated from a simple counter program (see Figure 10).

```
UINT4 count:
CHAN OF UNIT4 d:
PLACE d: AT "d0", "d1", "d2", "d3", ORDY "dor", IRDY "dir":
SEQ
  count := 5
  WHILE TRUE
    SEQ
      count := count PLUS 1
      d ! count
```

Figure 10 Initial OCCAM counter program

Another implementation (CTIM1) is taken from Peter Welch's commstime benchmark [P.H. Welch, 1988], with the sink process on channel 'd' removed to allow the channel to be placed onto the external interface (see Figure 11).

```
UINT4 count:
CHAN OF UNIT4 d:
PLACE d: AT "d0", "d1", "d2", "d3", OR "dor", IR "dir":
CHAN OF UINT4 a, b, c:
PAR
  SEQ -- prefix and pass-on
    a ! 5
    UINT4 v:
    WHILE TRUE
      SEQ
        b ? v
        a ! v
    SEQ -- fan-out
    UINT4 v:
    WHILE TRUE
      SEQ
        a ? v
        PAR
        c ! v
d ! v
    SEQ -- increment
    UINT4 v:
    WHILE TRUE
      SEQ
        c ? v
        b ! v PLUS 1
```

Figure 11 Single-value commstime program

Yet another implementation (CTIM2) uses a double-buffered commstime program to pass three values round in parallel (see Figure 12). This enables the developed circuit to output
values along channel ‘d’ at an improved rate. This is possible because the extra parallelism is implemented in hardware, while it would only be simulated by interleaving on a sequential processor.

\begin{verbatim}
UINT4 count:
CHAN OF UINT4 d:
PLACE d AT "d0", "d1", "d2", "d3", ORDY "dor", IRDY "dir":
CHAN OF UINT4 a, b, c:
PAR
  SEQ -- prefix and pass-on
    UINT4 x, y:
    SEQ
      a ! 5
      a ! 6
      b ? x
      a ! 7
    WHILE TRUE
    SEQ
      PAR
        b ? y
        a ! x
      PAR
        b ? x
        a ! y
  SEQ -- fan-out
    UINT4 v:
    WHILE TRUE
    SEQ
      a ? v
    PAR
      c ! v
      d ! v
  SEQ -- increment-by-three
    UINT4 v:
    WHILE TRUE
    SEQ
      c ? v
      b ! v PLUS 3
\end{verbatim}

Figure 12: Triple-value commutative program

The different size and structure of the generated logic influences the amount of state the CSP representation generates (see Table 1), as covered in [Peel & Pizarro 2005]. More parallelism creates a larger state space, but with the potential for better performance on the physical hardware.

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>( \text{CSP}_m ) Code Size (bytes)</th>
<th>Process Size (states)</th>
<th>Trace Refinement Time on 3.0 GHz P4 (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC</td>
<td>CTR</td>
<td>147k</td>
<td>16 + 876</td>
<td>3</td>
</tr>
<tr>
<td>SPEC</td>
<td>CTIM1</td>
<td>388k</td>
<td>16 + 1972</td>
<td>23</td>
</tr>
<tr>
<td>SPEC</td>
<td>CTIM2</td>
<td>511k</td>
<td>16 + 2418</td>
<td>62</td>
</tr>
<tr>
<td>CTIM1</td>
<td>CTIM2</td>
<td>870k</td>
<td>1972 + 2418</td>
<td>93</td>
</tr>
</tbody>
</table>

Table 1: Model-Checking Performance of FDR
3.8 Conclusion

As was indicated previously (see section 3.6.1 and section 3.7), even though this method of proving the generated hardware works, the proof is susceptible to state space explosion which becomes more apparent as the size and complexity of the application being converted increases. Even though techniques exist that can help minimise this problem, these techniques help to delay the problem, not solve it. This realisation, along with the fact that the proof strategy is viable if the circuits are kept small, has led onto the main body of the work in chapter 4 where we attempt to avoid the state space explosion by proving the process of creating the logic circuits.
4 Main Work

4.1 Premise for Work

Chapter 3 discussed a viable technique for modelling and proving small circuits which becomes infeasible as the circuit size increases. It was deemed more appropriate to devise a method to avoid the state space explosion rather than trying to ease or delay this problem. Since the generated circuits are composed of logic components, there exist patterns and structures within them. Identification of these patterns and structures would enable one to simplify the proof of that logic circuit. Rather than focussing on trying to search for and identify properties and patterns that exist by analysing the circuits, as the circuits have been designed automatically (as opposed to evolved), the task of proving the hardware compilation process of the compiler lends itself ideally to providing this structural information.

As logic circuits are being generated from software specifications (written in OCCAM), the initial specifications are definitively defined. The grammar and syntax of the OCCAM language limits both the number of unique component types and how these components can be combined together and thus interact. With the starting point for the software to hardware conversion containing structural information, it makes sense to utilise this source instead of trying to search for and extract the information after the logic circuit has been generated, which is what the technique in chapter 3 outlined.

Through utilising structural information derived from the OCCAM language, it is possible to simplify the CSP model of the corresponding circuit by replacing clusters of internally modelled logic components with a CSP model that performs an identical task but which contains less state space. Although this would enable the proof of correctness of larger and more complex circuits, this is not what this work attempts to do. To do so would have too large an impact on the end user of our compiler; some of the main implications and restrictions that this might introduce would be:

- Every generated logic circuit generated by the compiler having to be individually checked, a time consuming task.
• The number and size of models needing to be checked would increase as the size and complexity of the software that is converted to hardware increases. Thus the technique would be delaying any state space explosion instead of avoiding it.

• The user would be forced to write a CSP specification for each application that they wish to convert into hardware, as the proof technique requires the comparison of the modelled hardware against a specification.

Instead, as indicated, what this work does is to prove the method of generating the logic circuits and not the individually generated hardware circuits that are output. This is done by verifying the conversion of the individual OCCAM components into small segments of logic, proving that their behavioural properties cover the function that they are supposed to perform, while also ensuring that they can interact properly with all grammatically allowed components. The mapping of the OCCAM grammar into segments of logic circuit means, if one combines instances of these logic circuit segments together so that they represent the OCCAM software code through a one-to-one mapping, the generated hardware is guaranteed to perform as specified.

Proving the process of generating the logic circuits from a software specification is possible, because the problem of ensuring that the generated circuit performs as specified is completely independent from the problem of proving that the software specification achieves a sensible task. It is only possible to determine if a software specification achieves a sensible task if sufficient details of the problem it is trying to tackle are known. This thesis documents the design, development and proofs of a compiler tool, so it is impossible to know the exact details of what the compiler will be asked to compile. This means that proving that the generated logic circuits perform a sensible task is completely outside the scope of this project. As stated previously, what we can prove and guarantee is that the generated hardware will perform as specified. Through proving the generated hardware matches the specification, without any impact on whether the application performs a sensible task, it provides the end user to freely select how (if at all) they may wish to prove and determine the validity and correctness of his application. Proving that the application tackles the end user's desired task can be done independently of the proofs for the hardware generation, thus this can be done in whichever manner they deem most appropriate, i.e. they are not forced to use CSP.
4.1.1 Implementation Implications of Premise

The preliminary work covered in chapter 3 and Appendix A builds upon and integrates with the OCCAM to logic circuit compiler developed by Dr R. M. A Peel [Peel & Cook, 2000]. The logic circuit generation strategy utilised in that paper concentrates on creating optimised logic circuits that run efficiently without clearly defined boundaries between the segments of logic used in the conversion of the OCCAM software into hardware. This lack of clearly defined boundaries results from the compiler generating optimisations in the logic generation process at the interconnection boundaries for the OCCAM components. This stylistic choice for the logic generation process complicates the proof that the logic produced is always valid, due to the possibility that the OCCAM components may produce different logic circuit segments depending on how they are interconnected. However it does increase the performance of the generated circuits.

Through enforcing a more modular and constrained logic generation strategy, the aim of this research has been to produce a compiler where the conversion of OCCAM components into segments of logic circuit can be individually and independently checked and verified. Although this strategy may initially produce less efficient and/or larger amounts of logic, it should still be beneficial as the circuits generated by the compiler would not need to be checked or proven. The use of peephole optimisations could then be applied to a fully generated logic circuit. These optimisations could be allowed to cross the interconnecting boundaries between segments of logic for the OCCAM components, assuming the optimisations have been proven not to alter the behavioural properties of the circuit. Also, pre-processing of the software application being converted, so that more specialised segments of logic could be substituted where specific properties within the application can be identified, might provide avenues that would enable the generated logic to be better optimised.

The difference in the focus of the generation strategies that underpin the original compiler meant that it was deemed more effective to prototype the work in this chapter by developing a new compiler rather than retrofitting the new proof strategies onto the existing code. Although I chose to redevelop the compiler, the technical information and insight gained from the initial compiler development was easy to integrate into this new version.
4.2 Assumptions made within the Compiler and Proof Strategy

This section covers the assumptions made within the compiler design in this thesis:

- The abstract syntax tree (AST), that the compiler generates the hardware from, is assumed to be a correctly constructed representation for the target OCCAM application.
  - A non-formally-proven lexical tokeniser and syntax parser has been created at the front-end of the compiler. This automatically generates the AST from an application's OCCAM source code. It was built to simplify the parsing of larger examples, instead of hand generating the AST.
  - The structure of the AST is a direct one-to-one mapping of the application's OCCAM source code, so it is realistic to assume that the OCCAM can be parsed correctly and converted into a valid AST.

- The netlist data structure created by the compiler (representing the generated logic circuit), is assumed to be correctly output to a file that is used by the FPGA place-and-route configuration tools.
  - Both the compiler's internal logic circuit data structure and the output file represent the same netlist. The logic components and their interconnections in his netlist map in a bi-directional one-to-one fashion between these formats, and this is easy to verify.

- It is assumed that the tool, FDR2, that has been used to automatically check the generated proofs, performs correctly.
  - This a well established and stable tool, commonly used in the field of formal methods.

- The choice of FPGA clocking frequency is sufficiently low to enable the circuit to stabilise between clock cycles.
  - This operational condition underpins every clocked digital circuit. The FPGA development tools determine the maximum tolerable clock speed for each circuit.

These assumptions only affect the context within which the compiler operates, and do not detract from the assertions made in this chapter regarding its proof of correctness.
4.3 Project Overview

This project focused on the production of an OCCAM compiler that generates fine grained parallel clocked logic circuits, along with a formal proof for the generation of these logic circuits. This proof guarantees that the hardware generated performs as specified by the software specification, but does not prove that the software to be converted performs a sensible task; to do the latter would require detailed knowledge of why the application was written.

To avoid the state space explosion that is experienced when attempting to prove that a circuit satisfies its behavioural specification as the size and complexity of the application increases, this compiler utilises an object-oriented/modular approach. Generic super type specifications have been created describing all the allowable behavioural properties that the different implementable OCCAM components can perform; these are then used for two distinct but interconnected purposes.

The first purpose is to test and prove that segments of logic that are used to represent the OCCAM components in hardware are refinements of their corresponding super type specification (e.g. in Figure 13 the segment of logic circuit for an "IF THEN ELSE" component is a refinement of a "Control Flow Process", sharing the same interface structure of start, clock and reset inputs and a finish output). This is required because the second use for the generic specifications is as place fillers for internal components that are required for a component to function. If the component being examined requires other internal components to function, as in the "IF THEN ELSE" component seen in Figure 13 needing a boolean component and a "THEN" and "ELSE" control flow process component, the model of the segment of logic circuit can be connected to the corresponding generic specifications (e.g. the three yellow highlighted components in Figure 13). These generic specifications are used as place fillers to provide the full range of allowable behaviours that a correctly behaving internal component may perform. Some of the generic specifications require that internal choice be utilised to enable them to present all these possible behaviours.
Provided the segment of logic circuit for a component can be proven to be a refinement of its super type, as long as any internal components are behaving correctly, then it can be connected anywhere where its super type component may be used. This leads to a proof of composition, enabling the OCCAM component to be connected together in a one-to-one manner that directly mimics the grammatical structure of the software specification. Although this defines and proves how the components fit together, it does not guarantee that an individual OCCAM component performs its corresponding conceptual task (i.e. whether a segment of logic for an IF component performs an IF statement), just that it is the right shape.
Having a proof of composition, the remaining task is to prove that the segment of logic circuit for each OCCAM component performs its required conceptual task. This is achieved through the addition of extra events to the model, by running a CSP process in parallel with it. These events annotate and depict at a software level what is occurring, assigning meaning for specific occurrences of low level signals. These annotation events are added in two distinct locations, the first by annotating the outer level of the CSP model of the segment of logic circuit of each OCCAM component, annotating its interface signals (signals that are used to trigger and drive the logic, along with the signals that it returns). The second part is by adding extra events to the generic specifications, describing the conceptual software behaviour that this component performs at its outer boundary. The combination of these annotation events results in describing conceptually at a software level how the segment of logic circuit being tested triggers and interacts with any internal components that it utilises. Through hiding the annotation events, one is left with a CSP model that is behaviourally identical to the proof of composition model that was previously used, whereas hiding all the events other than the annotations produces a model that describes at a conceptual level what the segment of logic does. This conceptual model can then be compared with a hand crafted specification that describes the desired and expected behaviour of each component, proving their equivalence.
It is through the combination of the proof of composition and the proof that each component performs its relevant behavioural task, that the generated logic circuits can be guaranteed to perform as specified without the need to have their behaviour exhaustively checked against a behavioural specification. This makes use of CSP's congruence properties for these components. Although, for small circuits, the size and complexity of the proof would be greater than that of modelling and behaviourally checking the circuit as a whole, the main benefit arises because each OCCAM component only has to be checked once before it can be used in the compiler (and then it can be used repeatedly in future generated circuits). This guarantees that all circuits which can be generated by the compiler will perform as the source code specifies, thus avoiding the state space
explosion when the software to be converted into hardware increases in size or complexity.

4.4 Logic Component Representation

4.4.1 Why Continue Using CSP?

There are several reasons for continuing to use CSP in the main body of this work. Apart from building upon the preliminary work that utilises CSP (see chapter 3), several other factors contributed to the continued utilisation of CSP.

As the compiler proofs guaranteeing that the generated logic circuits perform as indicated by the OCCAM software are self contained (i.e. the hardware generation process is proven independently of any application that is converted), they can be completely isolated from any tests, models or proofs that the end user may decide to perform on his application. Although this compartmentalisation of the hardware generation proof from any proofs of the software application, that in itself was not a reason to utilise CSP, it does enable the choice of using CSP to have no impact or restriction upon how an end user may wish to prove or test their application. Thus enabling the selection of whichever formal language or proof methodology is desired, if the end user wishes to test or prove their application.

CSP is a formal event-based algebra which enables the formal specification of how parallel processes interact. As CSP is particularly suited to modelling processes, it can equally be used to represent the same problem in varying degrees of abstraction. As we can utilise CSP to represent models of specification in various levels of abstraction (both hardware and software descriptions), there was no compelling reason to select a different formal technique.

Selecting an alternative to CSP would either have caused the preliminary work covered in chapter 3 to have to be redone or the creation of a formal proof to convert between different formal methods. Both of these tasks would have been extra work that was deemed not critical to the solution of the problem.
4.4.2 Design Considerations & Implications

This thesis focuses on proving the construction process of generated logic circuits. The modular approach underpinning the chosen strategy requires that models of the logic components be sufficiently robust to accommodate features of the circuits that may not be guaranteed, desired or expected. Some of the features stated below, such as if a logic component is connected to itself, should never happen as the compiler has been designed not to produce this condition. Although this is the case, the CSP models must also be robust enough to deal with this because if a bug did exist within the compiler that produced this, the CSP models and proofs must be able to represent it. This is also the reason why this section of the work reintroduced the reset functionality back into the CSP models (to be able to detect if the reset pins are connected incorrectly by the compiler), as the work covered in chapter 3 removed it to try to help minimise the state space that had to be examined. As such the model of the logic components used in chapter 3 (specified in Appendix B), are not robust enough for the requirements of this section of the work. For example, the model of a D-Type Flip-Flop (see Figure 16), cannot deal with all the conditions specified below (i.e. if the output Q is connected to the input D, the model will not behave correctly).

```plaintext
PROC_ACTIVE_LIB_PDC(d, q, clock) =
let
  S(y) =
    d?x -> q!y -> c?1 -> S(x)
  []
    q!y -> d?x -> c?1 -> S(x)
within S(0)
```

*Figure 16: CSP model of D-Type Flip-Flop used in chapter 3*

Although modelling each logic component can be perceived as an individual task, because instances of these models are combined together to represent a circuit or segment of circuit, this imposes conditions that affect the design and implementation choices which were made.

- A signal (representing a wire connecting logic components) may be connected to multiple pins of a logic component; thus the models have to be designed to deal with this feature.
  - The model of the component should be comprised of multiple processes run in alphabetised parallel. This would enable common events in the
model that represent the signals received on the input and output pins of a component to synchronise.

- An alternative in some situations would be the application of peephole optimisations that could alter the circuit to remove any connected pins on a logic component without adversely affecting the overall behaviour of the circuit.

- Loops of logic containing only combinatorial (non-clocked) logic should not be allowed, as loops of non-clocked logic have the possibility of introducing race conditions.

  - If a loop is created and the possibility of a race condition exists, its occurrence would be detected by the specific run-time instantiation of the circuit (i.e. the specific signals being transmitted through the circuit).

  - Non-clocked logic components are modelled as IO-SEQ, and models containing loops of only these components are liable to deadlock. To ensure that this occurrence can be detected, by forcing the models to deadlock in FDR, the special case has to be considered where the loop consists of only one IO-SEQ component (i.e. it is wired to itself). To ensure that this will deadlock, the model of the component has to explicitly check that its outputs (e.g. output Q of a flip-flop) is not connected to its own inputs (e.g. input D of the same flip-flop), and explicitly deadlock if this does occur. Clocked logic components modelled as IO-PAR are not susceptible to deadlocking if they are connected in loops, but the individual IO-PAR CSP models will also have to check to see if they are connected to themselves. This is required to enable their corresponding inputs and outputs to be synchronised together in FDR so they are modelled correctly.

- Only one driving pin should be connected to any signal (wire).

  - If an attempt is made to drive a signal from multiple components or I/O pins, contention could occur. This would only cause problems in a hardware circuit if the different driving pins were trying to propagate different signal states at the same time. If the signals that they are trying to drive never differ from each other, then the hardware should not have a
problem, although it is likely in this case that the logic is redundant and thus may be optimised or removed.

- Running the processes representing the logic components in alphabetised parallel on the CSP events that represent the signal names causes them to deadlock if the processes attempt to drive different values. If the behaviour of the circuit is such that multiple drivers of a signal always drive the same state as each other, the CSP model of the logic circuit will be deadlock-free. This condition is impossible to test for as CSP allows ‘many-to-many’ channel synchronisations, and multiple drivers that continuously provide the same output do not cause a problem.

- Clocked logic components that the compiler uses only alter their output states on a low-to-high clock transition (e.g. D-Type Flip-Flops do this):
  - The clock can be modelled as a single event per cycle representing a clock low-to-high transition, thus helping to minimise the size of the state space required to be checked for the proofs, as covered in chapter 3 and [Peel & Pizarro 2005].

### 4.4.2.1 Combinatorial Logic

The components that fall under the IO-SEQ category (see section 2.6) are non-clocked combinatorial logic functions (AND, OR, NOT, etc). IO-SEQ is particularly suited to these components, because in any single clock period, outputs of these components are directly related to the inputs received in that clock cycle. Figure 17 demonstrates how to create an instantiation of the non-clocked IO-SEQ component (an AND gate), the relevant channels are parsed as arguments to the process specified in Figure 18 that defines the behaviour.
CSP Specification of an AND Gate

-- Declaration of signal channels that the gate
-- is wired to.
STATE = {0, 1}
channel chan_input1 : STATE
channel chan_input2 : STATE
channel chan_output : STATE

-- Declaration of an instantiation of a two
-- input AND gate.
INST_AND =
AND_GATE( chan_output,
          <chan_input1, chan_input2>
        )

-- Alphabet of instantiated AND gate
ALPHA_INST_AND =
[| chan_output, chan_input1, chan_input2 |]

Figure 17: An Instantiation of a Two Input AND Gate

-- Declaration of a channel used internally by the CSP specification. FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}

-- Instantiations of an AND_GATE calls this process and provides channels used
-- for its inputs and outputs as arguments. 'out' is a single channel to output
-- to. 'in_list' is a list of channels to input from.
AND_GATE(out, in_list) =
  let
    -- Check if this AND gate has any inputs.
    A =
      length(in_list) == 0 & B
    []
    length(in_list) == 0 & C
    -- Specification of an AND gate with no inputs, it behaves as GND.
    B = out 0 -> B
    -- Check if the output of the AND gate feeds any of its inputs, deadlock the
    -- specification if it does.
    C =
      inter( {out}, set(in_list) ) == {out} & STOP
    []
    inter( {out}, set(in_list) ) == {} & D
    -- AND gate specification.
    D =
      let
        -- Manage the inputs of the gate.
        DA =
          let
            -- Inputting process managing a single pin.
            -- 'x' becomes the input channel to manage.
            DAA(x) =
              x? y -> internal_state! y -> out? _ -> DAA(x)
            -- Run the inputs for all the input pins in parallel.
            DAB(x) =
              length(x) == 1 & DAA(head(x))
            []
            length(x) > 1 &
            ( DAA(head(x))
              [ | union(
                  DAA(head(x))
                [ | inter( {head(x)}, set(tail(x)) ),
                  [ | out ]
                ) ]
              )
            )
            DAB(tail(x))
within DAB(in_list)
-- Manage the output of the gate.
DB =
  let
  DBA =
    internal_state?1 -> DBC
    []
    internal_state?0 -> DBB
  DBB =
    out ! 0 -> DBA
    []
    internal_state?_ --> DBB
  DBC =
    out ! 1 -> DBA
    []
    internal_state?1 -> DBC
    []
    internal_state?0 -> DBB
within DBA
-- Connect the inputs and outputs together.
DC =
-- The ordering that the 'internal_state' events occur does not
-- matter, so long as they all occur. This is guaranteed by the
-- structure of the internal processes running in parallel.
( DA
  [] [ [ internal_state, out | ] ]
DB
) \ [ [ internal_state | ]
within DC
within A

Figure 18: AND Gate CSP Model: IO-SEQ
4.4.2.2 Clocked Logic

IO-PAR can represent clocked logic (D-Type Flip-Flops, T-Type Flip-Flops, etc). For any individual clock cycle, the output values are pre-determined and independent of the input values (although the input values will have an effect on the output value for the following clock cycle). It is because of this that these components can transmit their output values in parallel with receiving values on their inputs.

CSP Specification of a D-Type Flip-Flop

```csp
-- Declaration of signal channels that the gate
-- is wired to.
STATE = {0, 1}
channel chan_clock : {1}
channel chan_reset : STATE
channel chan_input : STATE
channel chan_output : STATE

-- Declaration of an instantiation of a
-- D Type Flip Flop.
INST_D =
  D_TYPE_FLIP_FLOP_IOPAR (  
    chan_clock, chan_reset,  
    chan_input, chan_output  
  )

-- Alphabet of instantiated D Type Flip Flop
ALPHA_INST_DTYPE =
  [{  
    chan_clock, chan_reset,  
    chan_input, chan_output  
  }]
```

A Graphical Depiction

```
Figure 19: An Instantiation of a D Type Flip Flop
```
-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}
channel internal_reset_state : {0, 1}

-- Instantiations of a D_TYPE_FLIP_FLOP calls this process and provides channels
-- used for its inputs and outputs as arguments. 'q_out' is a single channel to
-- output to. 'd_in' and 'reset' are single channels to input from. 'clock' is
-- an event representing a low-to-high clock transition.
D_TYPE_FLIP_FLOP_IOPAR(clock, reset, d_in, q_out) =
let
  -- Connect the inputs and outputs together.
  A =
    -- The initial default output value for this object is '0', hence the
    -- initial default value is passed to the process as an argument parameter
    -- 'C(0)'.
    -- The ordering that the 'internal_state' and 'internal_reset_state'
    -- events occur does not matter, so long as it occurs. This is guaranteed
    -- by the structure of the internal processes running in parallel, and has
    -- been verified by the assertion at the end of this example.
    ( let
      -- Run the inputs in parallel.
      BA =
        ( let
          -- Reset input
          BB = reset ? z -> internal_reset_state ! z -> clock -> BB
          -- D input
          BC = d_in ? z -> internal_state ! z -> clock -> BC
        )
        within BA
      -- Manage the inputs of the gate.
      let
        -- Manage the output of the gate. 'x' is the current state to output.
        C(x) =
          q_out ! x ->
          ( internal_reset_state ? 1 -> internal_state ? z -> clock -> C(0)
            []
            internal_reset_state ? 0 -> internal_state ? z -> clock -> C(z)
          )
        within A
    )
      C(0)
  )
\ { { internal_state, internal_reset_state |}

-- Verify that the ordering of the 'internal_state' and 'internal_reset_state'
-- does not matter
channel test0
channel test1, test2, test3 : {0, 1}
D_TYPE = D_TYPE_FLIP_FLOP_IOPAR(test0, test1, test2, test3)
assert chase(D_TYPE) (FD= D_TYPE

Figure 20: D-Type Flip-Flop CSP Model: IO-PAR
4.4.2.3 Adapting IO-PAR to OI-SEQ

To help to minimise the state space produced when modelling segments of logic, an adaptation of IO-PAR was produced. This is where the outputs are performed before the inputs and then followed by the clock, whereby the clock event acts as a barrier synchronisation ensuring that all components progress from one clock cycle to the next. This basic definition of the OI-SEQ concept can be seen in Figure 21, whereas Figure 22 is the definition reworked for the use of channels.

Although the utilisation of a barrier synchronisation event is not needed to ensure deadlock freedom, it is needed to be able to check if the clock has been correctly connected throughout the circuit. The utilisation of the clock as a barrier synchronisation, for all IO-PAR components, helps to minimise the state space that FDR2 has to check. The state is minimised because any IO-PAR components that may temporarily run ahead of others (as stated in the IO-PAR definition, theorem and proof covered in [P.H. Welch, 1987]), are forced to progress from one cycle to the next, in unison with all the other IO-PAR components. So long as the clock event is unique and not dependent on any of the other events or signals within the system, the definition, theorem and proof of the IO-PAR components covered in [P.H. Welch, 1987] is not invalidated. As previously mentioned, the barrier synchronisation event will ensure that the IO-PAR components progress from one cycle to the next in unison (which also happens to be the purpose of the clock signal within the circuit). This should also explain the position and utilisation of the clock event in the models used in [Peel & Wong, 2004] that this research builds upon.

Figure 21: A simple CSP definition of an OI-SEQ component

```
-- E = a set of all possible IO events
-- I = a set of input events
-- O = a set of output events
-- s = a barrier synchronisation event (e.g. a clock)
--
-- where
-- (not member(s,E)) and (inter(I,E) = I) and (inter(O,E) = O) and
-- (empty(inter(I,O))

OISEQ = (|| x:O o (x-> SKIP)); (|| x:I o (x-> SKIP)); s -> OISEQ
```

Figure 21: A simple CSP definition of an OI-SEQ component
- $E$ is a set of all possible IO channels
- $I$ is a set of input channels for a component
- $O$ is a set of output channels for a component
- $s$ is a barrier synchronisation event (e.g. a clock)
- $K$ is a set of default output events

where the following conditions hold true:

- The input channels for a component are not in its output channels
  \[ \text{empty}(\text{inter}(I, O)) \]
- The input and output channels are valid IO channels
  \[ \text{inter}(I, O) = I \text{ and } \text{inter}(O, E) = O \]
- The default output events are a set of events containing one event from the
  expansions of each of the components output channels
  \[
  K \leftarrow \{ \{a | a \leftarrow O \} \},
  \text{Card}(X) = \text{Card}(O),
  b \leftarrow X,
  c \leftarrow O,
  \text{member}(b, \{ | c | \}),
  d \leftarrow \text{diff}(X, \{ b \}),
  \text{not member}(d, \{ | c | \})
  \]

\[
\text{OISEQ} = \text{OISEQ2}(Z)
\]

\[
\text{OISEQ2}(A) =
\begin{align*}
&x:A \odot (\text{channel}(x) \land \text{message}(x) \Rightarrow \text{SKIP}); \\
&y:I \odot (x?_x \Rightarrow \text{SKIP}); \\
&s \rightarrow (\neg Y: \{ X \mid X \leftarrow \text{Set}(\{ | a | a \leftarrow O \}) \}, \\
&\text{Card}(X) = \text{Card}(O),
&b \leftarrow X,
&c \leftarrow O,
&\text{member}(b, \{ | c | \}),
&d \leftarrow \text{diff}(X, \{ b \}),
&\text{not member}(d, \{ | c | \})
\end{align*}
\]

**Figure 22:** A CSP definition of an OI-SEQ component using channels

As the output states for a clocked component are static during a specific clock cycle (i.e. they are not dependent on that cycle's input values), modelling the outputs occurring before the inputs helps to reduce the state space that has to be checked because the number of permutations for the OI-SEQ interleaving order is fewer than for the IO-PAR model (the IO-PAR in Figure 23 is replaced by an OI-SEQ, as shown in Figure 24).

**Figure 23:** IO-SEQ and IO-PAR connected in a loop
Figure 24: OI-SEQ replaces IO-PAR in IO-SEQ/IO-PAR loop

An example of a D-Type Flip-Flop modelled in the OI-SEQ form can be seen in Figure 25.
Main Work

-- Declaration of a channel used internally of the CSP specification CSP will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}
channel internal_reset_state : {0, 1}
channel internal

-- Instantiations of a D_TYPE_FLIP_FLOP calls this process and provides channels
-- used for its inputs and outputs as arguments. 'q_out' is a single channel to
-- output to. 'd_in' and 'reset' are single channels to input from. 'clock' is
-- an event representing a low-to-high clock transition.
D_TYPE_FLIP_FLOP_OISEQ(clock, reset, d_in, q_out) =
  let
    A =
      -- The ordering that the 'internal_state' and 'internal_reset_state'
      -- events occur does not matter, so long as it occurs. This is guaranteed
      -- by the structure of the internal processes running in parallel. The
      -- initial default output value for this object is '0', hence 'C(0)'.
      ( \B
        | union( \( \| clock,
                  internal,
                  internal_state,
                  internal_reset_state
        |\),
          \( \{ q_out \}, \{ d_in, reset \} \))
      )
    C(0)
  \{ \{ internal, internal_state, internal_reset_state \} \}

-- Manage the inputs of the gate.
B =
  let
    -- Run the inputs in parallel.
    BA =
      ( BB
        | union( \( \{ internal, clock \},
                  \( \| d_in \}, \| reset \})
        )
      )
    BC

-- Reset input
BB = internal -> reset ? z -> internal_reset_state ! z -> clock -> BB
-- D input
BC = internal -> d_in ? z -> internal_state ! z -> clock -> BC
within BA

-- Manage the output of the gate. 'x' is the current state to output.
C(x) = q_out : x -> internal ->
  ( internal_reset_state ? 1 -> internal_state ? _ -> clock -> C(0)
    |)
  internal_reset_state ? 0 -> internal_state ? z -> clock -> C(z)
within A

Figure 25: D-Type Flip-Flop CSP Model: OI-SEQ
Figure 26 demonstrates using FDR2, that the behaviour of the OI-SEQ component is a valid trace refinement of an IO-PAR version. Thus demonstrating that the OI-SEQ behaviour is contained within the IO-PAR behaviour.

---

--- E = a set of all possible IO events
--- I = a set of input events
--- O = a set of output events
--- s = a barrier synchronisation event (e.g. a clock)
---
--- where
--- (not member(s,E)) and (inter(I,E) = I) and (inter(O,E) = O) and
--- (empty (inter (I,O)))
---
--- IOPAR = ( || x:union(O, I) @ (x -> SKIP)); s -> IOPAR
--- OISEQ = ( || x:O @ (x -> SKIP)); ( || x:I @ (x -> SKIP)); s -> OISEQ
---
--- Check that the OISEQ model is a refinement of the OIPAR model
assert IOPAR [T= OISEQ

---

Figure 26: Assertions demonstrating OI-SEQ is a refinement of IO-PAR

OI-SEQ is useful for the specific style of clocked logic circuits that the compiler generates as it helps to minimise the state space that FDR has to check by reducing the number of interleaved orderings for parallel events, compared with the equivalent use of IO-PAR. The condition for utilisation of OI-SEQ imposes a constraint that no loops consisting of only these clocked logic components may exist to avoid the circuit model deadlocking. This is acceptable because the compiler never generates these loops, this condition may be verified because the component models are defined to behave as STOP if they detect that their outputs are directly connected to their inputs, and an FDR check can confirm that such a deadlock does not occur. For any OI-SEQ only loops consisting of more than one component, the CSP models will naturally deadlock, it is only the special case of loops of one component that has to be catered for.

4.5 Proof Framework

The proof of correctness of the compiler is complex, and involves many individual stages. These can be broken down into three interrelated phases.

1. The first phase is a proof that individual OCCAM constructs, when converted into segments of logic, can be connected together correctly in all possible combinations as indicated by the OCCAM grammar - i.e. correct composition.

2. The second phase is to prove that the logic for each of the individual types of OCCAM components performs its corresponding conceptual task.
3. The third and final phase ensures that the creation of the logic circuit by the compiler conforms to that which is indicated by the proof.

Examples of the various models and assertions used to demonstrate properties of points 1 and 2 can be found in Appendix D and Appendix E. These two appendixes respectively show the models and checks that need to be performed for a single type super type generic component and a single type implemented component.

There are currently six super type OCCAM components in our compiler; flow-control constructs (such as SEQ, PAR, IF or channel communications), data-read constructs (such as PLUS and numerical constants), data-store constructs, channel-read constructs, channel-send constructs and boolean constructs (such as TRUE, FALSE or less than). The reason that there are six types is due to the differing structural composition of their boundary connectors. There are also specialised multi-types (see section 4.5.4) - e.g. channels and variables - that provide access to two different types of interface constructs (e.g. ‘channel-read and channel-send’ or ‘data-read and data-send’ respectively); this is because they perform a mixture of these behaviours.

Instances of constructs can utilise components of any type internally to enable them to perform the required task. For example, the “IF THEN ELSE” component utilises two “Control Flow Processes” and a “Boolean Condition” as shown in Figure 13 on page 8. Proving that these components can be composed together requires generic CSP specifications to be written that depict the range of behaviours that are allowed to be performed at the outer boundaries of the super type OCCAM components. These specifications have two uses. For any specific OCCAM component that is being checked, if it requires other OCCAM components internally, then a corresponding super type generic specification can be used as the internal component. This enables the model of the implementable logic to determine all the legal possible states that it could ever get into, thus enabling a proof that it is a valid refinement of its super type generic component. Through proving that the implemented component is a valid refinement of a generic super type component, so long as any internal components used are also a valid refinement of their type of component, it is possible to fit all combinations of the implemented components together without invalidating or having to rerun any proofs. This demonstrates that the hardware for the OCCAM components can be fitted together in a manner that directly mimics the grammar of the language, because the logic of a verified
OCCAM component can be placed wherever its generic super type specification could be used (a property that is true because of congruence).

Although this proves that the composition of the logic circuits will be performed correctly, further work has to be done to demonstrate that the logic performs the correct high level behavioural task (e.g. that an "IF THEN ELSE" component when triggered performs a Boolean test, with the result of the test determining which of the "THEN" or "ELSE" blocks are triggered). To achieve this, extra annotation events have been added to the specifications to describe the higher level meanings of the states that the signals are in. Extra events were chosen to be added due to complications with achieving the same result through selective renaming (see Appendix J). These annotation events can then be compared by FDR against specifications that depict the desired and expected behaviours. Performing these tests enables the determination at a higher conceptual level what sequence of actions a component will perform and what conditions that these are dependent on. This enables us to demonstrate that an implemented component will correctly perform the high level behavioural task that it was designed to achieve (e.g. an 'IF THEN ELSE' block performs an 'IF THEN ELSE').

The combination of proving each individual implemented OCCAM component performs the task that its grammar component represents, along with proving that the implemented components can be connected together in a manner dictated by the OCCAM grammar, has the resultant implication that the components will interact in a way that is dictated by the software application code being converted into hardware. This provides a guarantee that the hardware will perform the task that the initial software application dictates, without having to check each individual circuit that the compiler could generate.

4.5.1 Appendix D Overview – A Single-Type Super-Type Component

This section details an overview of Appendix D, describing the CSP models it contains and their relationship with each other. Figure 27 illustrates these relationships graphically, representing a single, single-type super-type component. These specifications are designed to be utilised in the FDR model-check proofs described in Appendix E and section 4.5.2.
Figure 27: Overview of the relationship between the numerous models of an OCCAM supertype
Appendix D.1.1 contains an OI-SEQ specification that specifies the relationship between its external control signals (i.e. clock, reset, start and finish). External choice is used to allow for the different orderings of the start and reset signals, whereas internal choice is used to provide the specification with the freedom for a finish signal to be generated zero or more clock cycles after a start signal is received. This specification forces the correct sequencing of these signals.

Appendix D.1.2 provides a similar specification to D.1.1, but where incorrect values are allowed to be provided to the components inputs. The CSP specification then explicitly deadlocks if any of these invalid inputs are encountered, thus resulting in the failure of any FDR deadlock checks that this component is utilised in where this occurs.

Appendix D.1.3 is a process that when run in parallel with a model, enforces the correct input signals. This specification is similar to that presented in D.1.1, but now the internal choice that determined the state of the finish signal, is replaced with an external choice.

Appendix D.1.4 is a process similar to D.1.2, but containing extra semantic annotation events. These annotation events label the states that the process can take, labelling the outputs and the appropriate and inappropriate inputs. Hiding the annotation events results in a process that is failures divergence equivalent to D.1.2 (as covered in appendix D.2.6).

Appendix D.1.5 provides a process that when run in parallel with a model, adds outer level semantic annotation events that label the states that the process performs. The annotation events occur after the corresponding low level hardware events, although Appendix H (specifically H.1.5) provides an alternative, whereby the semantic annotation events precede the corresponding low level signal events.

Appendix D.1.6 contains a model built of only semantic annotations. These annotations represent the allowed states that can occur at a clock cycle based level (i.e. the NOTFINISHED state reflects one clock cycle in the hardware implementation).

Appendix D.4.1.1 provides a model of semantic annotations, ignoring clock cycles. This model gives a representation of the component that directly represents the software level of abstraction (e.g. when a control flow process is started, it may or may not finish).

Appendix D.2 contains the assertions that have to be proved for a super-type component. This proves various properties of the models and their relationships with each other, thus enabling the models to be used in the proofs contained in Appendix E.
Appendix D.2.1 provides a deadlock freedom check for the model in D.1.1.

Appendix D.2.2 utilises trace refinement to prove that the behaviour of D.1.1 is contained within D.1.2. The trace refinement is only performed in one direction, as this allows for D.1.2 to contain extra behaviour than D.1.1.

Appendix D.2.3 uses trace refinement to demonstrate that D.1.3 does not introduce extra behaviour to a process it is run in parallel with, but it may limit that processes events.

Appendix D.2.4 uses a deadlock freedom check to prove that D.1.2 is correctly limited by D.1.3 (through running them in parallel). Success in this check illustrates that the process is correctly driven, as the explicitly defined deadlocks in D.1.2 are never triggered.

Appendix D.2.5 utilises a pair of failures divergence refinements to ensure that D.1.2 limited by D.1.3 (through running then in parallel), has precisely the same behaviour as that defined in D.1.1.

Appendix D.2.6 is a failures divergence equivalence check that proves that the model in D.1.4, with its annotation events hidden, behaves identically to the model in D.1.2.

Appendix D.2.7 provides a trace refinement illustrating that D.1.3, does not add any extra behaviour to the model contained in D.1.4.

Appendix D.2.8 uses a deadlock freedom check to prove that D.1.3 limits the input events of D.1.4, so that the explicitly defined deadlocks do not occur.

Appendix D.2.9 contains a bidirectional failures divergence equivalence check, proving that if the annotation events are hidden from D.1.4, it behaves identically to D.1.2. This illustrates that the model in D.1.4 is identical to D.1.2, but contains semantic annotation events.

Appendix D.2.10 provides a deadlock freedom check for the clock cycle semantic annotation model contained in D.1.6.

Appendix D.2.11 is a bidirectional trace refinement used to prove that D.1.5 annotates a correctly performing process (e.g. D.1.1), such that the sequence of generated annotation events are the same as D.1.6.

Appendix D.2.12 contains a bidirectional failures divergence equivalence check. This check proves that D.1.5 only adds annotation events to a process it is run in parallel with (e.g. D.1.1), neither adding nor constraining the interface behaviour.
Appendix D.2.13 uses a failure divergence refinement check and a trace refinement check to illustrate that the annotation events contained within D.1.4 generate the same sequence of semantic annotations as D.1.6.

Appendix D.4.1.2 contains a failure divergence refinement check and a trace refinement check that links the clock cycle semantic annotation model (i.e. D.1.6), to the software semantic annotation model (i.e. D.4.1.1).

4.5.2 Appendix E Overview – A Single-Type Implemented Component

This section details an overview of Appendix E, describing the CSP models it contains and their relationship with each other and to the component’s super-type. Figure 28 illustrates these relationships graphically, representing a single, single-type implemented component. These specifications are the CSP proofs required to verify a low level hardware component.

Appendix E.1.1 specifies all the allowable signal transitions of the implemented low level hardware components outer interface (e.g. the clock, reset, start and finish signals). The chosen example is a component that implements an IF-THEN-ELSE statement.

Appendix E.1.2 is similar to D.1.2. Its basis is the E.1.1 specification, but it has been altered so that invalid inputs are allowed, with the incorrect inputs triggering explicitly defined deadlocks.

Appendix E.1.3 is an automatically generated model of the hardware created by the compiler. It is comprised of a number of logic gates whose inputs are joined together by CSP events, directly mimicking the logic net-list that represents the circuit. Figure 78, on page 8, illustrates graphically this component’s circuit.

Appendix E.1.4 shows a clock cycle based semantic annotation model of this component. It demonstrates how it drives the BOOLEAN check and the THEN and ELSE processes, and also how they interact and affect each other every clock cycle.

Appendix E.4.1.1 illustrates the semantic annotation model of this component.

Appendix E.2 contain the FDR assertions that link the various CSP specifications of the implemented component together, and to its super-type specifications covered in Appendix D.
Appendix E.2.1 performs a deadlock freedom check of the model of the circuit (E.1.3), with its inputs limited to being correctly driven (through running D.1.3 in parallel). Firstly, this checks that there are no loops of non-clocked logic, as this would result in the CSP model of the logic circuit deadlocking. Secondly, the deadlock freedom check guarantees that the logic circuit correctly drives any internal components. Failure to drive an internal component correctly will trigger the explicitly defined deadlock within the model, thus illustrating if the condition is violated.
Appendix E.2.2 uses a trace refinement to ensure that running the circuit model in parallel with D.1.3 (thus limiting its inputs so it is correctly driven), guarantees that the specification in D.1.3 does not add any extra behaviour to the circuit.

Appendix E.2.3 provides a deadlock freedom check, indicating that the boundary behaviour of the component is deadlock free.

Appendix E.2.4 uses a trace refinement to prove that the component is a valid sub-type of its super-type.

Appendix E.2.5 has a bidirectional failure divergence refinement check, that proves that when the implemented component is driven correctly, its interface behaviour is identical to the expected interface behaviour specified in E.1.1.

Appendix E.2.6 demonstrates that the hardware component annotated with the D.1.5 specification, does not deadlock.

Appendix E.2.7 is a deadlock freedom check on the high level clock cycle annotation only specification (i.e. E.1.4).

Appendix E.2.8 uses trace refinement to ensure that the implemented component behaves similarly to the allowed high level annotation specification.

Appendix E.4.1.2 contains trace refinement check that demonstrates that the annotation specification of E.4.1.1 is a valid sub-type of D.4.1.1.

4.5.3 Alternative Single-Type Component Models

An example of an alternative style of annotating a single-type component can be found in Appendix H and Appendix I. These appendices are similar to Appendix D and Appendix E, but with the difference being that the control process that is used to ensure the models are correctly driven when being analysed, uses internal choice to provide a valid input as opposed to limiting the allowed inputs of the models. The method for annotating the outer level of the models has also been adapted, whereby the semantic annotation events have been added to the control process and occur before the corresponding signal events. This adaptation results in the annotation only models having a more sequential style (e.g. an IF statement starts before its Boolean check), as opposed to the previous method where several annotation events are triggered in parallel (e.g. the IF statement and its Boolean check start in parallel in the same clock cycle).
Appendix J and Appendix K illustrate how much simpler the CSP models become if the reset functionality is not introduced into the proof framework. These appendices also demonstrate how removing the reset functionality, in this particular example, results in the semantic annotation events corresponding to specific instances of events (as opposed to a combination of specific instances of events), and how the process of connecting low level hardware models to semantic annotation models can be simplified.

4.5.4 Special Multi Type Components

The proof strategy covered in section 4.5 can be used to prove nearly all of the OCCAM components, with the exception of variables and channels (i.e. the multi type components). The reason that they have to be handled differently is because channels and variables perform two distinct functions; data can be assigned to a channel for transmission along it and also can be received from it, whereas variables can be written to and read from. These objects therefore maintain internal state that the proof models must accommodate, which is further complicated by the fact that an implementation of one of these components can have zero or multiple instances of each of their interface functions.

The proof methodology for these multi-type components is based on that of the single-type components. Single-type specifications are generated for each of the different conceptual interface functions that they can perform, along with an extra specification that dictates how instances of these interface specifications interact and influence each other. This gives a single-type component specification to use as an internal component in any proofs for other single-type components, whilst also being able to prove that the implemented logic that governs the interactions between these interfaces is valid and behaves as expected. This enables us to prove that the correct conceptual task occurs when the interfaces are accessed in parallel or sequentially, while still demonstrating that each of the implemented interfaces is still a valid refinement of the interface specification. An example of the CSP models and assertions for a multi-type generic super type specification can be found in Appendix F, and an example of an implemented component is contained in Appendix G.
4.6 Proof/Compiler Integration

The basic concept of the integration between the proof and the compiler is that, apart from generating the logic circuits, the compiler also composes together the CSP models to generate the proof. Through the use of inheritance, interfaces and other standard object oriented principles in Java, the compiler was designed and built with the concept of tight integration with the model generation in mind. The use of interfaces enables the code of the compiler that connects together the fragments of logic circuits representing the OCCAM components also to connect fragments of logic circuits to other custom hand crafted components. This functionality was designed for use in the generation of the CSP models, enabling the fragments of logic circuit to be connected to super type generic specifications that are substituted as place fillers for any required internal components. This feature has the side effect that the compiler has the ability to cope with connecting fragments of logic circuits that it generates with other components that may not have been created by the compiler. This means that any circuits or components that have been created through other design and development strategies have a simple avenue for continued reuse and integration into the compiler.

The combination of inheritance and strong type checking enables the compiler to ensure that OCCAM components which require other internal components can only be provided with valid sub type components (supplied as method arguments). This results in the Java type checker enforcing that the compiler can only compose together components that the OCCAM language grammatically allows, thus enforcing the proof of composition is upheld. The use of inheritance also results in the proofs for the implementable components to be automatically checked against their correct corresponding super type specifications (i.e. the super type that they physically inherit from); this is because the CSP models obtained from the super type components are integrated into the automatic construction of the proofs. If an implementable OCCAM component inherits from an incorrect class, the CSP proof that the compiler generates will fail when checked. This is because the compiler utilises the class structure to select the models encoded into fragments of CSP in a similar way that it utilises the class structure to determine how it converts components into hardware, thus ensuring consistency between the proofs that are run through FDR and the logic circuits generated.
5 Results

The redeveloped compiler currently is approximately 2.4MB of Java, consisting of over 320 classes and 46000+ lines of code. It generates 4+ MB of CSP split over more than 70 files, all of which have been run through FDR. As indicated throughout this work, the focus has been on guaranteeing the generation process of the logic circuits, thus ensuring the circuit behaves as specified by its software specification. The examples that were compiled to test the compiler were implemented on an FPGA and worked first time. The timings have been measured from the hardware and verified against the expected values computed by stepping through the application and using the rules stated in Appendix L.

5.1 Example 1: Commstime – Version 1

The example that was used to test the compiler was commstime, as specified in Figure 29.

```
SEQ
CHAN OF UINT7 a:
CHAN OF UINT7 b:
CHAN OF UINT7 c:
CHAN OF UINT7 d:
PAR
SEQ
  UINT7 v1:
    a ! 1
    WHILE TRUE
    SEQ
      b ? v1
      a ! v1
SEQ
  UINT7 v2:
    WHILE TRUE
    SEQ
      a ? v2
      PAR
      c ! v2
      d ! v2
SEQ
  UINT7 v3:
    WHILE TRUE
    SEQ
      c ? v3
      b ! v3 PLUS 1
SEQ
  PLACED UINT7 v4 AT "v0", "v1", "v2", "v3", "v4", "v5", "v6":
    WHILE TRUE
      d ? v4
```

Figure 29: Commstime written in OCCAM for conversion into hardware
Results

/* Instantiate an empty logic circuit and the start and finish signals */
LogicCircuit cir = new LogicCircuit();
LogicNet start = new LogicNet();
LogicNet finished = new LogicNet();

/* Declare the channels used in the specification */
OccamChannel_NotPlaced a = new OccamChannel_NotPlaced(cir, 7, "occamChannelA");
OccamChannel_NotPlaced b = new OccamChannel_NotPlaced(cir, 7, "occamChannelB");
OccamChannel_NotPlaced c = new OccamChannel_NotPlaced(cir, 7, "occamChannelC");
OccamChannel_NotPlaced d = new OccamChannel_NotPlaced(cir, 7, "occamChannelD");

/* Declare the variables used in the specification */
FlipFlopStorage v1 = new FlipFlopStorage(7); v1.setLogicCircuit(cir);
FlipFlopStorage v2 = new FlipFlopStorage(7); v2.setLogicCircuit(cir);
FlipFlopStorage v3 = new FlipFlopStorage(7); v3.setLogicCircuit(cir);
WatchedFlipFlopStorage v4 = new WatchedFlipFlopStorage(7, new String[] {
    "v0", "v1", "v2", "v3", "v4", "v5", "v6"}); v4.setLogicCircuit(cir);

/* Instantiate the AST */
OccamSequence_Seq outerSEQ = new OccamSequence_Seq(new OccamProcess[] {
    new OccamProcess_SingleChannelDeclaration(a),
    new OccamProcess_SingleChannelDeclaration(b),
    new OccamProcess_SingleChannelDeclaration(c),
    new OccamProcess_SingleChannelDeclaration(d),
    new OccamParallel_Par(new OccamProcess[] {
        new OccamProcess_Output_Channel(a.getChannelSend(),
            new OccamExpression_UINT_Constant(7, 1)),
        new OccamProcess_SingleVariableDeclaration(v1),
        new OccamProcess_While{
            new OccamBooleanTrue(),
            new OccamSequence_Seq(new OccamProcess[] {
                new OccamProcess_SingleChannelDeclaration(b),
                new OccamProcess_SingleVariableDeclaration(v1),
                new OccamProcess_While{
                    new OccamBooleanTrue(),
                    new OccamSequence_Seq(new OccamProcess[] {
                        new OccamProcess_SingleChannelDeclaration(a),
                        new OccamProcess_SingleVariableDeclaration(v1),
                        new OccamParallel_Par(new OccamProcess[] {
                            new OccamProcess_Output_Channel(a.getChannelSend(),
                                new OccamExpression_UINT_Constant(7, 1)),
                            new OccamProcess_SingleVariableDeclaration(v1),
                            new OccamProcess_While{
                                new OccamBooleanTrue(),
                                new OccamSequence_Seq(new OccamProcess[] {
                                    new OccamProcess_SingleChannelDeclaration(c),
                                    new OccamProcess_SingleVariableDeclaration(v2),
                                    new OccamProcess_While{
                                        new OccamBooleanTrue(),
                                        new OccamSequence_Seq(new OccamProcess[] {
                                            new OccamProcess_SingleChannelDeclaration(b),
                                            new OccamProcess_SingleVariableDeclaration(v2),
                                            new OccamProcess_While{
                                                new OccamBooleanTrue(),
                                                new OccamSequence_Seq(new OccamProcess[] {
                                                    new OccamProcess_SingleChannelDeclaration(c),
                                                    new OccamProcess_SingleVariableDeclaration(v3),
                                                    new OccamProcess_While{
                                                        new OccamBooleanTrue(),
                                                        new OccamSequence_Seq(new OccamProcess[] {
                                                            new OccamProcess_SingleChannelDeclaration(c),
                                                            new OccamProcess_SingleVariableDeclaration(v4),
                                                            new OccamProcess_While{
                                                                new OccamBooleanTrue(),
                                                                new OccamProcess_SingleChannelDeclaration(d)
                                                              } }) }) }) });
    } }) }) }) });

/* Generate the logic circuit */
outerSEQ.generateLogic(cir, start, finished);

Figure 30: Commstime hand translated into the AST for the compiler

65
The specification was then hand translated into its corresponding AST, as illustrated in Figure 30 on page 8, followed by the method call to instantiate the compilation into hardware. This generated a logic net list structure contained within the “LogicCircuit” component which was then output as EDIF (see cd: ex1), although this net list structure can also be output as CSP to provide a full circuit specification if desired. The generated EDIF was then passed to the “place and route” tools and programmed onto the FPGA.

Figure 31: Trace through for initialisation phase of Figure 29 commstime
Figure 32: Trace through of 6 cycle cyclical loop (N+9 to N+15) for Figure 29 commstime
The generated hardware that represents the application is triggered by a high state held for a single clock cycle on the "LogicNet start" wire of the outer component. When this occurs the hardware is triggered, the application performs its initialisation and the first output value (i.e. data stored in v4) is generated after 9 clock cycles (see Figure 31), this is followed by a new value after every further 6 clock cycles (see Figure 32). If this block of hardware were to finish, which it does not because it contains a "while true" loop, the hardware would have generated a high state on the "LogicNet finish" wire, held for a single clock cycle. The "WatchedFlipFlopStorage v4" component represents a placed variable with its corresponding signals connected to output pads, thus enabling the state within the variable to be viewed from outside the circuit.

5.2 Example 2: Commstime – Version 2

Just like in section 3.7, the performance of commstime may be improved by refactoring the application. The specification covered in Figure 33, when converted into hardware (see cd: ex2) and triggered, outputs its first value after 10 clock cycles, the second after another 4 and the third after a further 6. Subsequent values are output in cycles of every 6 and then 5 further clock cycles.

This example utilises a hand crafted lexical tokeniser and syntax parser to automatically parse the OCCAM source code and generate the AST. Although the syntax parser takes advantage of various features to help gain confidence in its AST generation (a formally proven parser would be the ideal solution but is outside the scope of this project). As stated in section 4.6, the Java type checking ensures that the subcomponents that the AST is comprised of can only be composed together in a manner that the OCCAM grammar allows. Although this helps to ensure that the AST the compiler utilises as its initial starting point is grammatically allowed, it does not guarantee that it is free of parallel usage errors (as this is a property of the OCCAM application).
SEQ
CHAN OF UINT7 a:
CHAN OF UINT7 b:
CHAN OF UINT7 c:
CHAN OF UINT7 d:
PAR
SEQ
  UINT7 v1:
  UINT7 v5:
    a ! 1
    a ! 2
    b ? v5
    a ! 3
  WHILE TRUE
  SEQ
    PAR
      b ? v1
      a ! v5
    PAR
      b ? v5
      a ! v1
SEQ
  UINT7 v2:
  WHILE TRUE
  SEQ
    a ? v2
    PAR
    c ! v2
    d ! v2
SEQ
  UINT7 v3:
  WHILE TRUE
  SEQ
    c ? v3
    b ! v3 PLUS 3
SEQ
  PLACED UINT7 v4 AT "v0", "v1", "v2", "v3", "v4", "v5", "v6":
  WHILE TRUE
  d ? v4

Figure 33: Commstime passing three values round the loop
Figure 34: Trace through for initialisation of Figure 33 commstime
Figure 35: Trace 11 cycle cyclical loop (N+17 to N+28) for Figure 33 commstime
It should be noted that by optimising the channel communication logic, i.e. removing the delay in the channel output (see section L.6), it would be possible to construct the circuit that outputs its first value after 10 clock cycles and all subsequent values outputs every further 3 clock cycles (see Figure 36 and Figure 37). The initialisation stage could also be optimised, and this could be achieved by declaring channels and variables in parallel, but not in a PAR. This would reduce the time required for the initialisation phase and shorten the time taken until the first value is output.
Figure 36: Optimised trigger & completion step through for initialisation of Figure 33 commstime
5.3 Example 3: A Digital Clock Displayed on an LCD

This section presents a larger example application (see Appendix N). The application is an implementation of a digital clock, on which the time is displayed on a liquid crystal display (LCD). The clock has two input buttons that allow the displayed time to be altered.
The application runs on a Xilinx FPGA and performs a power on initialisation of a Sitronix ST7066U graphics controller connected to an LCD, configuring it to communicate via its 4-bit data interface mode, and instructing it where to display the relevant characters. The application also samples two buttons which can be used to alter the running clock time, whilst several internal parallel processes interact via channel communications.

The developed application was broken down into the following segments:

- Main
- LCD_PowerOnInit
- LCD_Nibble
- LCD_Delay
- LCD_DisplayTime
- DigitalClock
- DebounceHour
- DebounceMin

5.3.1 Main

The segment of code, commented as "Main", runs the LCD_PowerOnInit code, followed by the following in parallel: LCD_DisplayTime, DigitalClock, DebounceHour and DebounceMin. It also declares locally the channels that DebounceHour and DebounceMin use to communicate with the DigitalClock process, and the channels that DigitalClock uses to communicate with LCD_DisplayTime.
5.3.2 LCD_PowerOnInit

This segment of code contains the sequence of commands and the required time delay between them, to perform the power on initialisation of the LCD graphics controller. This initialises and sets the controller to communicate in its 4bit data-mode.

The process utilises channels to send the individual commands to the LCD_Nibble code to output them to the LCD graphics controller. The process also communicates with the LCD_Delay process, triggering it via channels so that the required delay allows the commands to execute correctly.

5.3.3 LCD_Nibble

This process contains an infinite (WHILE TRUE) loop, listening on channels for commands it is required send to the LCD graphics controller. When this process receives a command, it sets the corresponding data and command lines to send the correctly timed command to the LCD graphics controller. The process then waits, before listening on its channels for the next command to send. This is so that there is a correctly-timed delay between individual commands.

5.3.4 LCD_Delay

This process contains an infinite (WHILE TRUE) loop. The process listens on a channel, starting a timer when it is instructed to do so. When the timer completes, the process communicates this fact back along another channel, before listening for when to start the timer again.

5.3.5 LCD_DisplayTime

This process contains an infinite (WHILE TRUE) loop. The process repeatedly listens on several channels for the time that is required to be displayed on the LCD. When it receives the time, it proceeds to send the required sequence of commands to the the LCD_Nibble process to indicate the correct characters to output. The LCD_DisplayTime process communicates with the LCD_Delay process, inserting the correct delays between the commands.
5.3.6 DigitalClock

This process executes an infinite \texttt{while true} loop. The process continuously updates running time, incrementing it every second. Every half a second, the process outputs the current time to the \texttt{LCD\_DisplayTime} process, whilst also listening to channels from the \texttt{DebounceHour} and \texttt{DebounceMin} processes to indicate if the user is manually altering the time (and adjusting it correspondingly).

5.3.7 DebounceHour and DebounceMin

These two processes both contain an infinite \texttt{while true} loop. The processes perform the same task, differing only by which user input button they sample and the channel they use to communicate to the \texttt{DigitalClock} process. The processes perform a simple debouncing on the user input (connected from a push button) by indicating that the state is a stable high input, if it has remained high for a specified period of time (i.e. a specific number of clock cycles), otherwise the state is set as low.

5.3.8 Compiling to Hardware

The application was parsed by the compiler and the generated logic circuit contained over 104900 lines of EDIF (see CD: ex3). The logic circuit was then configured onto a Xilinx Spartan-3E FPGA development board and run, a photo of the running hardware can be see in Figure 39.

After correctly specifying the wiring between the FPGA and the LCD controller, this circuit executed correctly the first time that it was run, again demonstrating the robustness of the compiler.
Figure 39: Example 3: Digital Clock, running on an FPGA development board
6 Future Work & Conclusion

The current implementation of the compiler has several obvious areas where further work can be focused. These range from expanding the subset of the grammar that the compiler supports, optimising the segments of logic used within the compiler, and adapting the compiler to build asynchronous logic circuits. Although the compiler is a prototype demonstrating a proof of concept, it shows the validity and viability of proving the construction process of building logic circuits from a software specification. The compiler can take OCCAM programs, from a limited subset of OCCAM, and compile them down into hardware. This is achieved without the need to model or test the generated logic circuits, as the process of generating the circuits has been proven to be correct.

By guaranteeing that all of the generated logic circuits produced by the compiler will always behave as specified by the corresponding supplied software specification, the extra state and complexity that is unavoidably introduced due to the conversion of the software into fine grained parallel hardware, has effectively been isolated and removed. Thus the process of converting a software specification into hardware does not increase the difficulty or complexity that the end users would require for them to prove or verify their software specification. Although this was the main focus for the project, which has been successfully achieved, other benefits are also present, such as the ability to utilise software design methodologies to simplify and speed up the process of building hardware applications.
Appendix A  Other Preliminary Work

A.1 Occam Grammar Modifications

The work on the grammar involved two main areas, both of these in preparation for the AST (abstract syntax tree) parser extensions. These were the expansion of the subset of Occam grammar that the compiler supports, and modifications to the grammar to enable it to be LALR1, which is a requirement for utilising SableCC [Gagnon, 1998].

A.2 AST Transformer Extensions

With SableCC [Gagnon, 1998] being utilised to develop the parser and abstract syntax tree (AST), the expanded grammar required the AST transformer to be expanded to accommodate the new features. This enabled it to deal with the modified grammar and to perform the circuit generation for the newly introduced grammar components. The generated logic circuit is represented as an internal net-list structure that can be viewed (see A.3) and simulated (see A.4), but also converted into a format (e.g. EDIF) suitable for final output from the compiler to the Xilinx FPGA synthesis tools.

A.3 Graphical Circuit Visualiser

To assist in the manual debugging of the generated logic circuits, I constructed a visualiser that graphically depicts the internal logic circuit’s net-list data structure that the compiler generates. This is the data structure that is translated into the final format output (e.g. EDIF) by the compiler.

The visualiser provides a simple layout with the components of the circuit stacked vertically on the left hand of the screen, and the interconnections between them to their right (see Figure 40). The tool enables the user to reorder the layout of the graphical components (both the logic components and interconnections) manually or semi-automatically with the assistance of two heuristic algorithms that I have developed. These are described in section A.3.1.
A.3.1 Auto Layout Simplification

My main two heuristics attempt to simplify the graphical depiction of the circuit through altering the order that the components are presented on the screen, each with a slightly different preference. Both of them attempt to reduce the number of times the lines interconnecting the components cross over each other, but the second algorithm also utilises the information concerning which components are driving the signals propagated along the interconnections.

Both of the algorithms I have developed are designed specifically for the graphical layout used, and they both follow a similar format. First, they order the logic components, followed by ordering the ports on those components, finally ordering the interconnecting signals between the ports. It is this relative ordering that determines the positions that the components are drawn onto the screen.
A.3.1.1 Algorithm 1

Being a semi-automatic heuristic, the user must first select which is the starting logic component that should be at the top of the list (an input to the circuit, e.g. the clock, is usually a good place to start – see Figure 41).

![Figure 41 Example output from the first tidying algorithm]

The algorithm proceeds to build up the order of logic components. The method of selecting the order of logic components is derived from examining the interconnecting signals (see Figure 42). To start, an empty set of interconnecting signals and an empty list of logic components are created. As each logic component is added to the end of the list (a process initially started by adding the initial logic component the user selected), the set of interconnecting signals is updated to include all the signals connected to the ports of this logic component. The set is then ordered by the number of logic components that it interconnects, but which are not yet contained within the new list. Any signals that do not connect to at least one logic component that is not currently in the new list are removed from the set. The first element in the set (if there is one), is selected, a logic component not contained within the new list that the signal connects to is selected at random and is added to the end of the new list (causing this list of processes to repeat). This continues until the set of interconnecting signals is empty. When it is empty, if all of the logic
components are not contained within the new list, the algorithm selects one of the remaining components at random and adds it to the end of the list (causing the list of processes to repeat), otherwise the new list contains the new order of logic components to draw to the screen.

User selects logic components to start algorithm on

SETUP
- Create an empty LIST for logic components
- Create an empty SET for interconnecting signals

Add selected logic component to end of the LIST

Add the interconnecting signals connected to the selected logic component to the SET

Order the SET ascending based on the number of unique logic components they connect to, which are not currently in the LIST

Remove any interconnecting signals form the SET that do not connect to any logic components that are not currently in the LIST

From the first interconnecting signal in the SET, randomly select a logic component that the signal connects to which is not currently contained in the LIST

Are all the logic components in the circuit in the LIST

The LIST contains the new order of logic components

Figure 42 A graphical representation of how logic components are ordered for algorithm I
Table 2: Port ordering imposed by the Comparator for the tidying algorithms

<table>
<thead>
<tr>
<th>Position of Port1 in interconnecting signal</th>
<th>Top</th>
<th>Bottom</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare the vertical length of the interconnecting signals connected to the ports</td>
<td>Port1 &gt; Port2</td>
<td>Port1 &lt; Port2</td>
<td>Port1 &lt; Port2</td>
</tr>
<tr>
<td>Port1 &gt; Port2</td>
<td>Port1 &gt; Port2</td>
<td>Port1 = Port 2</td>
<td></td>
</tr>
<tr>
<td>Port1 &lt; Port2</td>
<td>Port1 &lt; Port2</td>
<td>Port1 &lt; Port2</td>
<td></td>
</tr>
<tr>
<td>Compare the number of ports each port connects to</td>
<td>Port1 &gt; Port2</td>
<td>Port1 &lt; Port2</td>
<td>Port1 &lt; Port2</td>
</tr>
<tr>
<td>Result of ((\text{Port2 Signal}) - (\text{Port1 Signal}))</td>
<td>+ve</td>
<td>Port1 &gt; Port 2</td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td>-ve</td>
<td>Port1 &lt; Port 2</td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Port1 = Port 2</td>
<td></td>
</tr>
</tbody>
</table>

Position of Port2 in interconnecting signal:

Port1 > Port2

Compare the vertical length of the interconnecting signals connected to the ports

Result of \((\text{Port1 Signal}) - (\text{Port2 Signal})\)

Port1 > Port2

Other

Port1 > Port2 | Port1 < Port2 | Port1 = Port 2
The next stage of the algorithm involves ordering the ports on each of the logic components. This is achieved through utilising an existing modified merge sort [JavaDoc 1.5.0 Comparator] supplied with a custom built comparator (see Table 2). The comparator is used in the sorting process to determine if or when it should swap the location of two objects.

The last stage of the algorithm is sorting the order of the interconnecting signals. This is done in a similar way to how ports were ordered with a custom Comparator (see Table 3), examining both the vertical length of the signals and the number of ports that each signal connects to.

<table>
<thead>
<tr>
<th>Compute result of ((\text{Signal1 vertical length}) - (\text{Signal2 vertical length}))</th>
<th>+ve</th>
<th>-ve</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal1 &gt; Signal2</td>
<td>Signal1 &lt; Signal2</td>
<td>Compare the number of ports that each signal connects to</td>
<td>((\text{Signal1 num}) - (\text{Signal2 num}))</td>
</tr>
<tr>
<td>+ve</td>
<td>-ve</td>
<td>Zero</td>
<td></td>
</tr>
<tr>
<td>Signal1 &gt; Signal2</td>
<td>Signal1 &lt; Signal2</td>
<td>Signal1 = Signal2</td>
<td></td>
</tr>
</tbody>
</table>

A.3.1.2 Algorithm 2

The second heuristic (see Figure 43) performs in a similar manner to algorithm 1, the difference being derived from the ordering of the logic components generated. As previously stated, this algorithm attempts to incorporate information concerning which components are driving the signals propagated along the interconnections. This is done by having an alternative comparator to sort the logic components.
Figure 43 An example output from the second tidying algorithm
User selects logic components to start algorithm on

**SETUP**
- Create an empty LIST for logic components
- Create two empty sets for interconnecting signals, A and B

Add selected logic component to end of the LIST

Add the interconnecting signals connected to the selected logic component to the sets.
Signals from the output/bi-direction ports get added to set A.
Signals from the input/bi-direction ports get added to set B.

Order the sets ascending based on the number of unique logic components they connect to, which are not currently in the LIST.

Remove any interconnecting signals from the sets that do not connect to any logic components that are not currently in the LIST.

From the first interconnecting signal in the set, randomly select a logic component that the signal connects to which is not currently contained in the LIST.

Is set A empty

No

Is set B empty

No

Are all the logic components in the circuit in the LIST

No

Select a random logic component not currently contained within the LIST

The LIST contains the new order of logic components

Yes

Yes

Figure 44: A graphical representation of how logic components are ordered for algorithm 2
Appendix A: Other Preliminary Work

The algorithm proceeds to build up the order of logic components. The method of selecting the order of logic components is derived from examining the interconnecting signals (see Figure 44). Two empty sets (A and B) of interconnecting signals and an empty list of logic components are created. As a logic component is added to the end of the list (a process initially started by adding the initial logic component that the user has selected), the sets of interconnecting signals are updated to include the signals connected to the ports of the logic component. If a port is an output or bi-directional port for a logic component, then the signal connected will be added to set A. If a port is an input or bi-directional port for a logic component, then the signal connected will be added to set B. The sets are then ordered by the number of logic components that they interconnect, but which are not yet contained within the new list. Any signals that do not connect to at least one logic component that is not currently in the new list are removed from the sets. The first element in the set A (if there is one), is selected, a logic component not contained within the new list that the signal connects to is selected at random and is added to the end of the new list (causing this list of process to repeat). If set A is empty, then the process is performed for set B. This continues until both sets of interconnecting signals are empty. When they are empty, if all of the logic components are not contained within the new list, the algorithm selects one of the remaining components at random and adds it to the end of the list (causing the list of processes to repeat), otherwise the new list contains the order new order of logic components to draw to the screen.

The algorithm then continues to order the ports and then the interconnecting signals, as covered in A.3.1.1.

A.4 Circuit Simulator

I have integrated a logic simulator with the visualiser (see A.3), that enables instances of the circuit to be run and tested. By providing a waveform for the input ports into the circuit as a CSV (Comma Separated Value) file (see Figure 45), the simulator can compute the results which are a set of waveforms for the circuit (see Figure 46).
Appendix A: Other Preliminary Work

Figure 45 A CSV file (shown in Excel) representing input waveforms to a circuit

Due to the fact that a simulator simulates a single instance of the circuit, and does not exploring all possible states it can get into, the method of simulation involves transmitting values along the interconnecting signals only if they change. When a value gets transmitted along a signal, this triggers any logic components connected to it to be simulated. The result of a simulated logic component can (if a value has changed) cause a transmission along an interconnecting signal resulting in the relevant logic components being triggered and simulated. This process helps to minimise the need for computing values and segments of the circuit that remain unchanged from one clock cycle to another.
The result of the simulator can be produced in various output formats. Apart from producing a CSV file (similar to the file used to specify the input waveforms), a graphical representation of the waveforms can be viewed (see Figure 46) or specific signals of the simulation can be viewed using the graphical visualiser (see A.3) with the high/low values of signals being depicted as red or black (see Figure 40, Figure 41 and Figure 43).
Appendix B  Logic Component CSP Models for Preliminary Work

This appendix contains the CSP models of the logic components utilised for the preliminary work covered in chapter 3.

B.1 AND Gates

B.1.1 AND Gate with 1-Input

PROC_ACTIVE_LIB_AND1 (o, i0) =
let
  I = i0?x -> o!x -> I
within I

B.1.2 AND Gate with 2-Inputs

PROC_ACTIVE_LIB_AND2 (o, i0, il) =
let
  O(a, b) =
    a = 1 and b = 1 & o!1 -> SKIP
    a = 0 or b = 0 & o!0 -> SKIP
  I =
    i0?x -> il?y -> O(x, y); I
    il?y -> i0?x -> O(x, y); I
within I

B.1.3 AND Gate N-Inputs

AND gates that contain three or more inputs currently are generated by utilising a tree of AND gates, an example of a three input AND gate can be seen in Figure 47.
The CSP for this three input AND gate example is shown below.

```
channel chan_active_lib_and3_a : {0, 1}
channel chan_active_lib_and3_b : {0, 1}

PROC_ACTIVE_LIB_AND3(o, i0, i1, i2) =
  let
    hid_alpha_active_lib_and3 =
      [! chan_active_lib_and3_a, chan_active_lib_and3_b ]
    S1 =
      PROC_ACTIVE_LIB_AND1(chan_active_lib_and3_a, i0)
    S2 =
      PROC_ACTIVE_LIB_AND2(chan_active_lib_and3_b, i1, i2)
    S =
      PROC_ACTIVE_LIB_AND2(o,
        chan_active_lib_and3_a,
        chan_active_lib_and3_b)
  within
    (S1 || S2) [! hid_alpha_active_lib_and3 ] S
  \ hid_alpha_active_lib_and3
```

### B.2 D-Type Flip-Flop

```
PROC_ACTIVE_LIB_FDC(d, q, c) =
  let
    S(y) =
      d?x -> q!y -> c?1 -> S(x)
    [!]
    q!y -> d?x -> c?1 -> S(x)
  within S(0)
```

### B.3 GND

```
PROC_ACTIVE_LIB_GND(ground) =
  let
    I = ground!0 -> I
  within I
```
Appendix B: Logic Component CSP Models for Preliminary Work

B.4 Inverter

PROC_ACTIVE_LIB_INV(i, o) =
  let
  I = i?x -> o!(1-x) -> I
within I

B.5 OR Gates

B.5.1 OR Gate with 1-Input

PROC_ACTIVE_LIB_OR1(o, i0) =
  let
  I = i0?x -> o!x -> I
within I

B.5.2 OR Gate with 2-Inputs

PROC_ACTIVE_LIB_OR2(o, i0, i1) =
  let
  O(a, b) =
    a == 1 or b == 1 & o!1 -> SKIP
    []
    a == 0 and b == 0 & o!0 -> SKIP
  I =
    i0?x -> i1?y -> O(x, y); I
    []
    i1?y -> i0?x -> O(x, y); I
within I

B.5.3 OR Gate with N-Inputs

OR gates that contain three or more inputs currently are generated by utilising a tree of three OR gates in a manner that is similar to that covered in section B.1.3. An example of a three input OR gate can be seen in Figure 48.
Appendix B: Logic Component CSP Models for Preliminary Work

The CSP for a three input OR gate is show below.

```csp
class channel chan_active_lib_or3_a : {0, 1}
class channel chan_active_lib_or3_b : {0, 1}

PROC_ACTIVE_LIB_OR3(o, i0, i1, i2) =
let
  hid_alpha_active_lib_or3 =
  { [ chan_active_lib_or3_a, chan_active_lib_or3_b ] } 
  S1 =
  PROC_ACTIVE_LIB_OR1(chan_active_lib_or3_a, i0)
  S2 =
  PROC_ACTIVE_LIB_OR2(chan_active_lib_or3_b, i1, i2)
  S =
  PROC_ACTIVE_LIB_OR2(o, chan_active_lib_or3_a, chan_active_lib_or3_b, chan_active_lib_or3_b)
within
(  (S1 || S2) [ ] hid_alpha_active_lib_or3 [ ] S  ) \ hid_alpha_active_lib_or3
```

B.6 VCC

```csp
PROC_ACTIVE_LIB_VCC(vcc) =
let
  I = vcc:1 -\rightarrow I
within I
```

B.7 XOR Gates

B.7.1 XOR Gate with 1-Input

```csp
PROC_ACTIVE_LIB_XOR1(o, i0) =
let
  I = i0?:x -\rightarrow o!x -\rightarrow I
```
B.7.2 XOR Gate with 2-Inputs

PROC_ACTIVE_LIB__ XOR2(o, i0, i1) =
let
  O(a, b) =
    a != b & o!1 -> SKIP
    []
    a == b & o!0 -> SKIP
  I =
    i0?x -> i1?y -> O(x, y); I
    []
    i1?y -> i0?x -> O(x, y); I
within I
Appendix C  Logic Component CSP Models for Main Work

The CSP models covered in this appendix are used to create the automatically generated CSP representation of the logic net list used in the proof covered in chapter 4. The processes defined here are instantiated (as shown in Figure 17 page 8 and Figure 19 page 8) and then run in parallel, thus mimicking the generated logic circuits with a one-to-one mapping.

C.1  Developing the Models

As stated in section 3.4, the reasons for adapting the models used in the work described in chapter 3, is covered in section 4.4.2. This section will provide an example of how several of the key points affect the design and construction of the models.

channel dff_out, dff_in : {0, 1}
channel clock

DFF(state) =
  dff_out!state -> dff_in?next -> clock -> DFF (next)
[]
  dff_in?next -> dff_out!state -> clock -> DFF (next)

DFLIPFLOP = DFF (0)

Figure 49: CSP Definition of a D-Type Flip-Flop from [Peel & Wong, 2004]

Figure 49 is a description of a d-type flip-flop, as defined in [Peel & Wong, 2004]. Figure 50 is the same model, packaged such that the supplied arguments are the channels that the process utilises. This packaging is also utilised in the models that are defined in this appendix, the following examples will demonstrate how the defined models may be constructed through incremental changes.

PROC__ACTIVE_LIB__FDC(d, q, c) =
 let
  S(y) =
    d?x -> q!y -> c?1 -> S(x)
  []
  q!y -> d?x -> c?1 -> S(x)
within S(0)

Figure 50: CSP Definition of a D-Type Flip-Flop from Appendix B, used in Chapter 3
With the definition of the flip-flop covered in Figure 50 not being able to function properly if the channels supplied for its input and output (d and q) are the same (e.g. the flip-flop is connected to itself, or after the reset is added to the model, the same channel is fed to both the reset and the d input). To fix this, there must be multiple internal processes running in alphabetised parallel, enabling them to synchronise if they share common events, but run interleaved when they do not. To achieve this correctly, an extra channels must be introduced, thus enabling the processes to interact and communicate. The extra events introduced are not and should not be identifiable outside the process representing the flip-flop, thus the newly introduced events are hidden. The let-within block enables the processes to be clustered logically together, as they represent newly created sub-processes of the flipflop, but as the let-within blocks within FDR2 will not allow one to define channels inside them, the introduced channels must be declared globally. These modifications can be seen in Figure 51.

```plaintext
channel internal_state : {0, 1}
PROC__ACTIVE_LIB_FDC (d, q, c) =
  let
    -- Run the internal processes in parallel
    FDC =
      ( FDC_D
          | [ [ d, c, internal_state | ] || [ q, c, internal_state | ] ]
          & [ [ internal_state | ] ]
        ) \ [ [ internal_state | ] ]
    -- Process the input
    FDC_D = d? x -> c? l -> internal_state? x -> FDC_D
    -- Process the output
    FDC_Q(x) = q? x -> c? l -> internal_state? y -> FDC_Q(y)
  within FDC
```

Figure 51: A Modified CSP Definition of a D-Type Flip-Flop – Version 1

As the newly introduced events that enable the internal processes to communicate are being hidden, furthermore, adding an extra input for the reset functionality produces a model that behaves as that described in section C.3.

Although the models covered in this appendix were initially based on those defined in [Peel & Wong, 2004], they were not in fact developed through incremental changes (as described above). The models were developed in a holistic manner, taking into consideration all the factors and concerns identified through an examination of the initial model, and then recreating the models in a single step to solve, tackle and deal with them.
Appendix C: Logic Component CSP Models for Main Work

C.2 AND

-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}

-- Instantiations of an AND_GATE calls this process and provides channels used
-- for its inputs and outputs as arguments. 'out' is a single channel to output
-- to. 'in_list' is a list of channels to input from.
AND_GATE(out, in_list) =
 let
  -- Check if this AND gate has any inputs.
  A =
  length(in_list) == 0 & B
  []
  length(in_list) != 0 & C
  -- Specification of an AND gate with no inputs, it behaves as GND.
  B = out ! 0 -> B
  -- Check if the output of the AND gate feeds one of the inputs, deadlock the
  -- specification if it does.
  C =
  inter( {out}, set(in_list) ) == {out} & STOP
  []
  inter( {out}, set(in_list) ) == {} & D
  -- AND gate specification.
  D =
  let
    -- Manage the inputs of the gate.
    DA =
    let
      -- Inputting process managing a single pin.
      -- 'x' becomes the input channel to manage.
      DAA(x) =
        x?y -> internal_state!y -> out?_ -> DAA(x)
      -- Run the inputs for all the input pins in parallel.
      DAB(x) =
        length(x) == 1 & DAA(head(x))
        []
        length(x) > 1 &
        { DAA(head(x))
          [ union(
            inter( {head(x)}, set(tail(x)) ),
            [{ out }]
          ), ]
        }
        DAB(tail(x))
    )
    within DAB(in_list)
    -- Manage the output of the gate.
    DB =
    let
      DBA =
        internal_state?1 -> DBC
      []
      internal_state?0 -> DBB
      DBB =
        out ! 0 -> DBA
      []
      internal_state?_ -> DBB
      DBC =
        out ! 1 -> DBA
      []
      internal_state?1 -> DBC
      []
      internal_state?0 -> DBB
    within DBA

98
Appendix C: Logic Component CSP Models for Main Work

-- Connect the inputs and outputs together.
DC =
-- The ordering that the 'internal_state' events occur does not
-- matter, so long as they all occur. This is guaranteed by the
-- structure of the internal processes running in parallel.
( DA
   [[ {{ internal_state, out }} ]]
DB
) \ {{ internal_state }}
within DC
within A

C.3 D-Type Flip-Flop

-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : (0, 1)
channel internal_reset_state : (0, 1)

-- Instantiations of a D_TYPE_FLIP_FLOP calls this process and provides channels
-- used for its inputs and outputs as arguments. 'q_out' is a single channel to
-- output to. 'd_in' and 'reset' are single channels to input from. 'clock' is
-- an event representing a low-to-high clock transition.
D_TYPE_FLIP_FLOP(clock, reset, d_in, q_out) =
  let
  -- Connect the inputs and outputs together.
  A =
  -- The ordering that the 'internal_state' and 'internal_reset_state'
  -- events occur does not matter, so long as it occurs. This is guaranteed
  -- by the structure of the internal processes running in parallel. The
  -- initial default output value for this object is '0', hence 'C(0)'.
  ( B
    [[ union( [] clock,
       internal_state,
       internal_reset_state
    ]],
     inter( [] q_out [], [] d_in, reset [] )
    ]
  )
  C(0)
) \ {{ internal_state, internal_reset_state }}

-- Manage the inputs of the gate.
B =
  let
  -- Run the inputs in parallel.
  BA =
    ( BB
      [[ union( [] clock []), inter( [] d_in [], [] reset [] ) ]]
       BC
    )
  -- Reset input
  BB = reset ? z -> internal_reset_state ! z -> clock -> BB
  -- D input
  BC = d_in ? z -> internal_state ! z -> clock -> BC
  within BA

-- Manage the output of the gate. 'x' is the current state to output.
C(x) =
  q_out ! x ->
  ( internal_reset_state ? 1 -> internal_state ? _ -> clock -> C(0)
   []
   internal_reset_state ? 0 -> internal_state ? z -> clock -> C(z)
  )
  within A
Appendix C: Logic Component CSP Models for Main Work

C.4 GND

-- Instantiations of a GND calls this process and provides the output channel -- for its output as an argument. 'out' is a single channel to output to.
GND(out) = out_0 -> GND(out)

C.5 Inverter (NOT)

-- Instantiations of an inverter (NOT_GATE) calls this process and provides -- channels used for its input and output as arguments. 'out' is a single -- channel to output to. 'in' is a single channel to input from.
NOT_GATE(out, in) =
let
-- Check if the output of the NOT gate feeds the input, deadlock the -- specification if it does.
A =
out := in & STOP
[]
out := in & B
-- NOT gate specification.
B = in ? z -> out ! (1-z) -> B
within A

C.6 NAND

-- Declaration of a channel used internally by the CSP specification FDR will -- not allow you to declare channels inside a let-within block, all channels -- must be globally defined.
channel internal_state : {0, 1}

-- Instantiations of an NAND_GATE calls this process and provides channels used -- for its inputs and outputs as arguments. 'out' is a single channel to output -- to. 'in_list' is a list of channels to input from.
NAND_GATE(out, in_list) =
let
-- Check if this NAND gate has any inputs.
A =
length(in_list) == 0 & B
[]
length(in_list) := 0 & C
-- Specification of a NAND gate with no inputs, it behaves as VCC.
B = out ! 1 -> B
-- Check if the output of the NAND gate feeds one of the inputs, deadlock -- the specification if it does.
C =
inter( {out}, set(in_list) ) == {out} & STOP
[]
inter( {out}, set(in_list) ) == {} & D
-- NAND gate specification.
D =
let
-- Manage the inputs of the gate.
DA =
let
-- Inputting process managing a single pin.
-- 'x' becomes the input channel to manage.
DAA(x) =
x?y -> internal_statey -> out?_ -> DAA(x)
-- Run the inputs for all the input pins in parallel.
DAB(x) =
length(x) == 1 & DAA(head(x))
Appendix C: Logic Component CSP Models for Main Work

\[
\begin{align*}
\text{DAA}(\text{head}(x)) \\
\text{union}
\quad \text{inter}(\{\text{head}(x), \text{set}(\text{tail}(x))\}, \\
\quad \{\text{out}\}) \\
\text{DAB}(\text{tail}(x))
\end{align*}
\]

within DAB(in_list)

-- Manage the output of the gate.

\[
\text{let}
\begin{align*}
\text{DBA} &= \text{internal state}\ ? 1 \rightarrow \text{DEC} \\
\text{DBB} &= \text{internal state}\ ? 0 \rightarrow \text{DBB} \\
\text{DBC} &= \text{out} ! 1 \rightarrow \text{DBA} \\
\text{DB} &= \text{internal state}\ ? _\rightarrow \text{DBB}
\end{align*}
\]

within DBA

-- Connect the inputs and outputs together.

\[
\text{DC} =
\begin{align*}
\text{let}
\begin{align*}
\text{DA} &= \text{internal state}, \text{out} \rightarrow \text{DEC} \\
\text{DB} &= \text{internal state}, \text{out} \rightarrow \text{DBB}
\end{align*}
\end{align*}
\]

within DC

\[
\begin{align*}
\text{channel internal state} : \{0, 1\}
\end{align*}
\]

C.7 NOR

-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.

channel internal state : \{0, 1\}

-- Instantiations of an NOR_GATE calls this process and provides channels used
-- for its inputs and outputs as arguments. 'out' is a single channel to output
-- to. 'in_list' is a list of channels to input from.

NOR_GATE(out, in_list) =

let

-- Check if this NOR gate has any inputs.
\[
\begin{align*}
\text{A} &= \text{length}(\text{in_list}) \leq 1 \& B \\
\text{B} &= \text{out} ! 0 \rightarrow B
\end{align*}
\]

-- Specification of an NOR gate with no inputs, it behaves as GND.

\[
\begin{align*}
\text{C} &= \text{inter}(\{\text{out}\}, \text{set}(\text{in_list})) \leq \{\text{out}\} \& \text{STOP} \\
\text{D} &= \text{inter}(\{\text{out}\}, \text{set}(\text{in_list})) \leq \{\} \& \text{D}
\end{align*}
\]
Appendix C: Logic Component CSP Models for Main Work

-- NOR gate specification.
D =
  let
  -- Manage the inputs of the gate.
  DA =
    let
      -- Inputting process managing a single pin.
      -- 'x' becomes the input channel to manage.
      DAA(x) =
        x?y -> internal_state!y -> out?_ -> DAA(x)
    -- Run the inputs for all the input pins in parallel.
    DAB(x) =
      length(x) == 1 & DAA(head(x))
      []
      length(x) > 1 &
      ( DAA(head(x))
        [ [ union(
            inter({head(x)}, set(tail(x))) ),
            [ [ out ]]
        ) ]
        DAB(tail(x))
      )
    within DAB(in_list)
  -- Manage the output of the gate.
  DB =
    let
      DBA =
        internal_state?1 -> DBB
        []
        internal_state?0 -> DBC
      DBB =
        out !0 -> DBA
        []
        internal_state?_ -> DBB
      DBC =
        out !1 -> DBA
        []
        internal_state?1 -> DBB
        []
        internal_state?0 -> DBC
    within DBA
  -- Connect the inputs and outputs together.
  DC =
    -- The ordering that the 'internal_state' events occur does not
    -- matter, so long as they all occur. This is guaranteed by the
    -- structure of the internal processes running in parallel.
    DA
    [ [ [ internal_state, out ] ] ]
  DB
  ) \ [ [ internal_state ] ]
within DC
within A

C.8 OR

-- Declaration of a channel used internally by the CSP specification PDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}

-- Instantiations of an OR_GATE calls this process and provides channels used
-- for its inputs and outputs as arguments. 'out' is a single channel to output
-- to. 'in_list' is a list of channels to input from.
OR_GATE(out, in_list) =
  let
Appendix C: Logic Component CSP Models for Main Work

-- Check if this OR gate has any inputs.
A =
    length(in_list) == 0 & B
    []
    length(in_list) != 0 & C
-- Specification of an OR gate with no inputs, it behaves as GND.
B = out ! 0 -> B
-- Check if the output of the OR gate feeds one of the inputs, deadlock the
-- specification if it does.
C =
    inter( {out}, set(in_list) ) == {out} & STOP
    []
    inter( {out}, set(in_list) ) == {} & D
-- OR gate specification.
D =
let
-- Manage the inputs of the gate.
DA =
let
    -- Inputting process managing a single pin.
    -- 'x' becomes the input channel to manage.
    DAA(x) =
        x?y -> internal_state?y -> out?_ -> DAA(x)
    -- Run the inputs for all the input pins in parallel.
    DAB(x) =
        length(x) == 1 & DAA(head(x))
        []
        length(x) > 1 &
            ( DAA(head(x))
            [] union( 
                inter( {head(x)}, set(tail(x)) ),
                [] out []
            ) []
            )
        DAB(tail(x))
    )
within DAB(in_list)
-- Manage the output of the gate.
DB =
let
    DBA =
        internal_state?1 -> DBB
        []
        internal_state?0 -> DBC
    DBB =
        out ! 1 -> DBA
        []
        internal_state?_ -> DBB
    DBC =
        out ! 0 -> DBA
        []
        internal_state?1 -> DBB
        []
        internal_state?0 -> DBC
    )
within DBA
-- Connect the inputs and outputs together.
DC =
    -- The ordering that the 'internal_state' events occur does not
    -- matter, so long as they all occur. This is guaranteed by the
    -- structure of the internal processes running in parallel.
    ( DA
        [] { internal_state, out [] } []
    ) \ { internal_state []
within DC
within A

103
C.9 T-Type Flip-Flop

-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : \{0, 1\}
channel internal_reset_state : \{0, 1\}

-- Instantiations of a T_TYPE_FLIP_FLOP calls this process and provides channels
-- used for its inputs and outputs as arguments. 'q_out' is a single channel to
-- output to. 't_in' and 'reset' are single channels to input from. 'clock' is
-- an event representing a low-to-high clock transition.
T_TYPE_FLIP_FLOP(clock, reset, t_in, q_out) =
  let
    -- Connect the inputs and outputs together.
    A =
      -- The ordering that the 'internal_state' and 'internal_reset_state'
      -- events occur does not matter, so long as it occurs. This is guaranteed
      -- by the structure of the internal processes running in parallel. The
      -- initial default output value for this object is '0', hence 'C(0)'.
      let
        (B
          [\] union( [\] clock, internal_state, internal_reset_state )
        ,
          inter( [\] q_out , [\] t_in , reset ) )
        )
      \{ C(0) \}
    -- Manage the inputs of the gate.
    B =
      let
        -- Run the inputs in parallel.
        BA =
          ( BB
            [\] union( [\] clock ), inter( [\] d_in , [\] reset ) )
          )
        BC
        -- Reset input
        BB = reset ? z -> internal_reset_state ! z -> clock -> BB
        -- T input
        BC = t_in ? z -> internal_state ! z -> clock -> BC
      within BA
    -- Manage the output of the gate. 'x' is the current state to output.
    C(x) =
      q_out ! x ->
      ( internal_reset_state ? 1 -> internal_state ? _ -> clock -> C(0)
        \[
          internal_reset_state ? 0 ->
            ( internal_state ? 1 -> clock -> C(1-x)
              \[
                internal_state ? 0 -> clock -> C(x)
              )
            )
        )
      within A
    
C.10 VCC

-- Instantiations of a VCC calls this process and provides the output channel
-- for its output as an argument. 'out' is a single channel to output to.
VCC(out) = out ! 1 -> VCC(out)
C.11 XOR

-- Declaration of a channel used internally by the CSP specification FDR will
-- not allow you to declare channels inside a let-within block, all channels
-- must be globally defined.
channel internal_state : {0, 1}

-- Instantiations of an XOR_GATE calls this process and provides channels used
-- for its inputs and outputs as arguments. 'out' is a single channel to output
-- to. 'in_list' is a list of channels to input from.
XOR_GATE(out, in_list) =
  let
    -- Check if this XOR gate has any inputs.
    A =
      length(in_list) == 0 & B
      []
    length(in_list) != 0 & C
    -- Specification of an XOR gate with no inputs, it behaves as GND.
    B = out ! 0 -> B
    -- Check if the output of the XOR gate feeds one of the inputs, deadlock the
    -- specification if it does.
    C =
      inter( {out}, set(in_list) ) == {out} & STOP
      []
    inter( {out}, set(in_list) ) == {} & D
    -- XOR gate specification.
    D =
      let
        -- Manage the inputs of the gate.
        DA =
          let
            -- Inputting process managing a single pin.
            -- 'x' becomes the input channel to manage.
            DAA(x) =
              x?y -> internal_state?y -> out?_ -> DAA(x)
            -- Run the inputs for all the input pins in parallel.
            DAB(x) =
              length(x) == 1 & DAA(head(x))
              []
              length(x) > 1 &
              ( DAA(head(x))
                [] union( inter( {head(x)}, set(tail(x)) ),
                out []
                )
              )
            DAB(tail(x))
            within DAB(in_list)
          -- Manage the output of the gate.
          DB =
            let
              DBA =
              out ! 0 -> DBA
              []
              internal_state?1 -> DBC
              []
              internal_state?0 -> DBA
              DBB =
              out ! 0 -> DBA
              []
              internal_state?_ -> DBB
              DBC =
              out ! 1 -> DBA
              []
              internal_state?1 -> DBB
              []
internal_state?0 -> DBC
within DBA

-- Connect the inputs and outputs together.
DC =

-- The ordering that the 'internal_state' events occur does not
-- matter, so long as they all occur. This is guaranteed by the
-- structure of the internal processes running in parallel.
( DA
  [] [[ internal_state, out ]] ]
DB
) \ [[ internal_state ]] 
within DC
within A
Appendix D   Single Type Component

Generic Specification Model Example

This appendix explains the CSP models required for a generic super-type component, along with the assertions that need to be checked to link the models to each other (see Figure 52). The models covered in this appendix describe in various levels of abstraction, the behaviours that an implementable OCCAM component (which is a sub type of this generic type) has to conform within (as covered in chapter 4), along with specifications dictating the correct driving behaviours that a sub component may have to deal with.
Appendix D: Single Type Component Generic Specification Model Example

Figure 52: Overview of the relationship between the numerous models of an OCCAM supertype
Appendix D: Single Type Component Generic Specification Model Example

D.1 Models & Specifications

The 'internalChoice' event that may appear within the code examples has been utilised instead of internal choice (i.e. '|-|') to enable 'chase' compression to be applied if desired. The 'internalChoice' event must be hidden for the specifications to be valid, but if 'chase' compression has been chosen, the event should only be hidden after 'chase' has been applied, otherwise the specification becomes invalid.

D.1.1 GenSpec 1: Valid Low Level Behaviour

This model specifies all the valid and allowable low level behaviour of this type of super type component. The purpose is to describe the interface boundary behaviours, thus enabling implemented components to refinement check against it proving their behaviours are within the requirements for it to be a sub-type of this super-type.

```plaintext
PROC_PROCESS_DESIRED_GENERIC_SPEC(clock, reset, start, finish) =
let
  A =
    start?x -> reset?y -> C(x, y)
    []
    reset?y -> start?x -> C(x, y)
  B =
    start?0 -> reset?y -> D(y)
    []
    reset?y -> start?0 -> D(y)
  C(x, y) =
    y == 1 & clock?1 -> finish!0 -> A
    []
    y == 0 &
      ( x == 0 & clock?1 -> finish!0 -> A
        []
        x == 1 & clock?1 ->
          ( internalChoice -> finish!1 -> A
            [] -- [-]
            internalChoice -> finish!0 -> B
          )
      )
    D(y) =
    y == 1 & clock?1 -> finish!0 -> A
    []
    y == 0 & clock?1 ->
      ( internalChoice -> finish!1 -> A
        [] -- [-]
        internalChoice -> finish!0 -> B
      )
  within finish!0 -> A
```

Figure 53: Low Level Generic Control Flow Specification

This specification (see Figure 53) will only accept correct input driving signals, and will return valid output result signals. Internal choice is utilised to enable it to specify all the possible valid refinements.
Appendix D: Single Type Component Generic Specification Model Example

D.1.2 GenSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 54) is based on the one covered in section D.1.1, but with the altered fact that it also accepts invalid driving input signals to be submitted to it. These invalid input driving signals are followed by an explicitly defined 'STOP', that will explicitly deadlock the model should it ever be reached. Similar to the specification in section D.1.1, the returned output signals will be all possible valid permutations allowed (internal choice is utilised to create those permutations, so long as it is driven correctly).

The reason why this model will accept invalid driving signals is to enable the model of any component connected to it the opportunity to provide any driving signals it may choose, this process will not limit or remove the possibility for the other component models to provide invalid signals to this one as an option when they are run in alphabetised parallel. The purpose of this is to enable possibility to check that if this specification is used as an internal component, so long as the outer component is driven correctly, this component will be driven correctly.
Figure 54: Low Level Generic Control Flow Specification with Explicit Deadlocking

D.1.3 GenSpec 3: Correct Component Driving

This CSP model is used to limit a process so that it can only accept possible valid input signals, this is to enable implemented sub-type components to have the outer layer of their logic correctly driven when performing the checks and proofs. The aim for this is to check an implemented component holds true to the assumption that so long as it is driven correctly, it will correctly drive any internal components.
Appendix D: Single Type Component Generic Specification Model Example

**Figure 55: Generic Control Flow Specification - Correct Driving Limiter**

### D.1.4 GenSpec 4: Annotated Low Level Behaviour with Explicit Deadlocking

This CSP model is the one covered in section D.1.2, but with extra events added to describe conceptually what is occurring. The aim of this is to enable a link between a low level hardware model and a higher level conceptual meaning of the function the hardware is performing. The added 'id' parameter added to the process is to provide a method to distinguish between different instances of this process. The annotation events depicting the states that are entered into from how this component is driven can only be specified after the event has occurred, where as the output signals are controlled by this component and so the corresponding annotation events can be performed before outputting the signals. The reason why renaming can not be used to obtain a higher level conceptual model of what is occurring, thus the required use of extra events depicting the annotations, is because the same signal states can mean different things depending on the state of the system (e.g. a start signal high state 'start?1' can mean that this component has been triggered, or that this component is being driven incorrectly and an error has occurred).
channel chan_controllFlowAnnotatedSpec : {1, 2}. {0, 1}
PROC_PROCESS_ANNOTATED_SPEC(clock, reset, start, finish, id) =
let
  A =
    start?x -> reset?y -> C(x, y)
    reset?y -> start?x -> C(x, y)
  B =
    start?l -> annotation.ERROR.id -> STOP
    start?0 -> reset?y -> D(y)
    reset?y ->
      start?0 -> D(y)
    start?1 -> annotation.ERROR.id -> STOP
  C(x, y) =
    y == 1 & annotation.RESET.id -> clock?1 -> finish!0 -> A
    x == 0 &
      ( x == 0 & annotation.IDLE.id -> clock?1 -> finish!0 -> A
        x == 1 & annotation.START.id -> clock?1 ->
          ( internalChoice -> annotation.FINISH.id -> finish!1 -> A
            [ ] [ ]
            internalChoice -> annotation.NOTFINISHED.id -> finish!0 -> B
          )
        )
    )
  D(y) =
    y == 1 & annotation.RESET.id -> clock?1 -> finish!0 -> A
    y == 0 & clock?1 ->
      ( internalChoice -> annotation.FINISH.id -> finish!1 -> A
        [ ] [ ]
        internalChoice -> annotation.NOTFINISHED.id -> finish!0 -> B
      )
  within finish!0 -> A

Figure 56: Annotated Generic Control Flow Specification with Explicit Deadlocking

D.1.5 GenSpec 5: Annotating the Outer Layer

This CSP model is used to annotate the outer layer of an implemented sub-type of this component. The process allows correct input and output signals to be able to annotate and describe that an error has occurred. No internal choice is used, as this should not restrict or control a process, but only annotate what is occurring.
Appendix D: Single Type Component Generic Specification Model Example

```
PROC_PROCESS_ANNOTATE OUTER(clock, reset, start, finish) =
let
  A =
    start?x -> reset?y -> C(x, y)
    []
    reset?y -> start?x -> C(x, y)

  B =
    start?1 -> annotation.ERROR.0 -> STOP
    []
    start?0 -> reset?y -> D(y)
    []
    reset?y ->
      (start?0 -> D(y)
       []
       start?1 -> annotation.ERROR.0 -> STOP
       )

  C(x, y) =
    y == 1 & annotation.RESET.0 -> clock?1 ->
      (finish!0 -> A
       []
       finish!1 -> annotation.ERROR.0 -> STOP
       )
    []
    x == 1 & annotation.START.0 -> clock?1 ->
      (finish!1 -> annotation.FINISH.0 -> A
       []
       finish!0 -> annotation.NOTFINISHED.0 -> B
       )
    []

  D(y) =
    y == 1 & annotation.RESET.0 -> clock?1 ->
      (finish!0 -> A
       []
       finish!1 -> annotation.ERROR.0 -> STOP
       )
    []
    y == 0 & clock?1 ->
      (finish!1 -> annotation.FINISH.0 -> A
       []
       finish!0 -> annotation.NOTFINISHED.0 -> B
       )
  within finish!0 -> A
```

Figure 57: Generic Control Flow Annotate Outer Layer

D.1.6 GenSpec 6: Clock Cycle Higher Generic Specification

This CSP model is an annotation only clock cycle based higher conceptual specification. It is used as a comparison for the extracted annotations from the annotated low level hardware models. The model is sufficiently small so that it is unlikely that ‘chase’ compression should be needed to be applied, which is why internal choice (i.e. ‘|’-) is used instead of using an extra event to simulate internal choice.
Appendix D: Single Type Component Generic Specification Model Example

D.2 Assertions: Linking the Models Together

To ensure consistency between all the models for a generic component, several assertions have to be proven. The consistency between the models is required because the proof of an implemented component can utilise several of these specifications.

D.2.1 GenSpec Assertion 1: Initial Deadlock Check

This initial deadlock-free check (see Figure 59) of GenSpec 1 (see section D.1.1) is to provide a base comparison for future deadlock-free checking and trace refinement.

---

```plaintext
@ channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the model to check
-- The alpha_PROC4 contains the low level channels used by this instance
alpha_PROC4 = [{ chan0, chan1, chan2, chan3 }]
PROC4 = PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that PROC4 has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )

-- Deadlock-free check the expected correct generic component
assert GEN_SPEC1 : [deadlock free [F]]
```

---

Figure 59: Example of GenSpec Assertion 1 for Control Flow Process
Appendix D: Single Type Component Generic Specification Model Example

D.2.2 GenSpec Assertion 2: GenSpec 2 Contains GenSpec 1 Behaviour

This assertion (see Figure 60) demonstrates that GenSpec 2 (see section D.1.2) contains all the behaviour dictated by GenSpec 1 (see section D.1.1), although this assertion allows GenSpec 2 to provide extra behaviours.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC3 = { | chan0, chan1, chan2, chan3 |
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

alpha_PROC4 = { | chan0, chan1, chan2, chan3 |
PROC4 = PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC2 = ( PROC3 \ {internalChoice} )

-- Check GenSpec 2 contains the behaviour of GenSpec 1
assert GEN_SPEC2 [T= GEN_SPEC1
```

Figure 60: Example of GenSpec Assertion 2 for Control Flow Process

D.2.3 GenSpec Assertion 3: GenSpec 3 Compatible with GenSpec 2

This assertion (see Figure 61) demonstrates that the GenSpec 3 controlling specification (see section D.1.3) does not introduce any new behaviour to the specifications it is being run in parallel with. This still leaves the possibility of it limiting the events that can occur, but does not guarantee any properties regarding this.
D.2.4 GenSpec Assertion 4: GenSpec 3 Removes Deadlock from GenSpec 2

This assertion (see Figure 62) demonstrates that GenSpec 3 (see section D.1.3) removes the possibility of driving the component it is run in parallel with, incorrectly. This does not dictate that GenSpec 3 does not remove a correctly driving option.
D.2.5 GenSpec Assertions 5: GenSpec 3 Removes Only Incorrect Driving

These assertions (see Figure 63) demonstrate that GenSpec 3 limits the process it is run in parallel with, such that it only allows correct driving signals. These assertions also demonstrate that GenSpec 3 does not introduce any extra behaviours and does not remove any correct driving options, it is achieved through proving that GenSpec 3 run in parallel with GenSpec 2 is indistinguishable to GenSpec 1.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC2 - (J chan0, chan1, chan2, chan3 |)
PROC2 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan3)

alpha_PROC3 = (| chan0, chan1, chan2, chan3 |)
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

alpha_PROC4 = (| chan0, chan1, chan2, chan3 |)
PROC4 = PROC_PROCESS_DESIVED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC2 = ( PROC3 \ {internalChoice} )
GEN_SPEC3 = PROC2

-- GenSpec2 limited by the control specification GenSpec3
GEN_SPEC2_WITH_CONTROLL = GEN_SPEC2 [] alpha_PROC2 ] GEN_SPEC3

-- Check that GenSpec3 in parallel with GenSpec2 is indistinguishable to GenSpec1
assert GEN_SPEC1 [FD= GEN_SPEC2_WITH_CONTROLL
assert GEN_SPEC2_WITH_CONTROLL [FD= GEN_SPEC1
```

Figure 63: Example of GenSpec Assertions 5 for Control Flow Process

D.2.6 GenSpec Assertions 6: Properties of the Annotation Events

These assertions (see Figure 64) demonstrate that the annotation event contained within GenSpec 4 do not introduce extra behaviours, but are only used to conceptually describe what is occurring. This is achieved through hiding the annotation events, and proving that the resultant process is indistinguishable to the GenSpec 2 model.
Appendix D: Single Type Component Generic Specification Model Example

-- channel declarations
data type STATES =
  START. {0} | FINISH. {0} | NOTFINISHED. {0} | RESET. {0} | ERROR. {0} | IDLE. {0}
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC0 = [ chan0, chan1, chan2, chan3 ]
PROC0 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)
alpha_PROC3 = [ chan0, chan1, chan2, chan3 ]
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = ( PROC3 \ {internalChoice} )
GEN_SPEC4 = ( PROC0 \ {internalChoice} )

-- GenSpec4 with the annotations hidden
GEN_SPEC4_NOANNOTATIONS = ( GEN_SPEC4 \ { annotations } )

-- Check that GenSpec3 in parallel with GenSpec2 is indistinguishable to
-- GenSpec1
assert GEN_SPEC4_NOANNOTATIONS [FD= GEN_SPEC2
assert GEN_SPEC2 [FD= GEN_SPEC4_NOANNOTATIONS

Figure 64: Example of GenSpec Assertions 6 for Control Flow Process

D.2.7 GenSpec Assertion 7: GenSpec 3 Compatible with GenSpec 4

This assertion (see Figure 65) proves that the control process GenSpec 3 does not add extra behaviours to the annotated generic specification GenSpec 4.

-- channel declarations
data type STATES =
  START. {0} | FINISH. {0} | NOTFINISHED. {0} | RESET. {0} | ERROR. {0} | IDLE. {0}
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC2 = [ chan0, chan1, chan2, chan3 ]
PROC2 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan3)
alpha_PROC0 = [ chan0, chan1, chan2, chan3 ]
PROC0 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = ( PROC0 \ {internalChoice} )

-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROL = GEN_SPEC4 [ \ alpha_PROC2 ]

-- Check GenSpec4 contains the behaviour of GenSpec4 limited by GenSpec3
assert GEN_SPEC4 [T= GEN_SPEC4_WITH_CONTROL

Figure 65: Example of GenSpec Assertion 7 for Control Flow Process
Appendix D: Single Type Component Generic Specification Model Example

D.2.8 GenSpec Assertion 8: GenSpec 3 Removes Deadlock From GenSpec 4

This assertion (see Figure 66) proves that the control process GenSpec 3 removes the explicitly defined deadlocks from the annotated generic specification GenSpec 4.

```plaintext
-- channel declarations
datatype STATES =
  START.{0} | FINISH.{0} | NOTFINISHED.{0} | RESET.{0} | ERROR.{0} | IDLE.{0}
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC2 = {chan0, chan1, chan2, chan3}
PROC2 = PROC_PROCESS_CONTROLL(chan0, chan1, chan2, chan3)

alpha_PROC0 = {chan0, chan1, chan2, chan3}
PROC0 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = (PROC0 \ {internalChoice})

-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROLL = GEN_SPEC4 \ alpha_PROC2 \ GEN_SPEC3

-- Check that GenSpec3 removes the incorrect driving options from GenSpec4
assert GEN_SPEC4_WITH_CONTROLL : [deadlock free [F]]
```

Figure 66: Example of GenSpec Assertion 8 for Control Flow Process

D.2.9 GenSpec Assertions 9: GenSpec 4 is an Annotated GenSpec 2

These assertions (see Figure 67) demonstrate that if the annotation events contained within GenSpec 4 are hidden, then GenSpec is indistinguishable to GenSpec 2.
Appendix D: Single Type Component Generic Specification Model Example

```
-- channel declarations
datatype STATES =
    START.{0} | FINISH.{0} | NOTFINISHED.{0} | RESET.{0} | ERROR.{0} | IDLE.{0}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha PROC contains the low level channels used by the processes
alpha PROC3 = { chan0, chan1, chan2, chan3 }
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

alpha PROC0 = { chan0, chan1, chan2, chan3 }
PROC0 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)
GEN_SPEC4 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)

-- GenSpec6 with hidden annotation events
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)

-- Check that GenSpec6 is deadlock-free
assert GEN_SPEC6 ; [deadlock free [F]]
```

Figure 67: Example of GenSpec Assertions 9 for Control Flow Process

D.2.10 GenSpec Assertion 10: GenSpec 6 Deadlock Free

This assertion (see Figure 68) deadlock-free checks GenSpec 6 which helps to provide a base comparison for future deadlock-free checking and trace refinement for the annotations.

```
-- Channel declarations
datatype STATES =
    START.{0} | FINISH.{0} | NOTFINISHED.{0} | RESET.{0} | ERROR.{0} | IDLE.{0}
channel annotation : STATES

-- Higher Outer Spec
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)

-- Check that GenSpec6 is deadlock-free
assert GEN_SPEC6 ; [deadlock free [F]]
```

Figure 68: Example of GenSpec Assertion 10 for Control Flow Process

D.2.11 GenSpec Assertions 11: GenSpec 5 Annotates Outer Level Correctly

These assertions (see Figure 69) demonstrates that GenSpec 5 will annotate the outer level of a correct process with annotation events that are similar to GenSpec 6. The
Appendix D: Single Type Component Generic Specification Model Example

assertions can be failures divergent checked (‘\([\text{FD}]=\)’) because the events that represent the low level signals which have been hidden, determine the initial state and thus the annotation that occurs.

```
-- channel declarations
datatype STATES =
  START.{0} | FINISH.{0} | NOTFINISHED.{0} | RESET.{0} | ERROR.{0} | IDLE.{0}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha PROC contains the low level channels used by the processes
alpha_PROC1 = [{ chan0, chan1, chan2, chan3 |}
PROC1 = PROC_PROCESS_ANNOTATE_OUTER(chan0, chan1, chan2, chan3)

alpha_PROC4 = [{ chan0, chan1, chan2, chan3 |}
PROC4 = PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC5 = PROC1
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)

-- GenSpec1 with annotation events added by GenSpec 5 running in parallel
GEN_SPEC1_WITH_ANNOTATIONS = ( GEN_SPEC1 \ alpha_PROC1 |] GEN_SPEC5 )

-- The annotations only that were added to GenSpec1 by GenSpec 5
GEN_SPEC1_ANNOTATIONS_ONLY = (GEN_SPEC1_WITH_ANNOTATIONS \ alpha_PROC4 )

-- Check that GenSpec4 with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC1_ANNOTATIONS_ONLY [T= GEN_SPEC6
assert GEN_SPEC6 [T= GEN_SPEC1_ANNOTATIONS_ONLY
```

Figure 69: Example of GenSpec Assertion 11 for Control Flow Process

D.2.12 GenSpec Assertions 12: GenSpec 5 Does Not Introduce Extra Behaviour

These assertions (see Figure 70) demonstrates that GenSpec 5 does not add extra behaviours, but only adds events that conceptually annotates the process it is run in parallel with. This is shown by the fact that the process run in parallel with GenSpec 5, with the annotations then hidden, is equivalent to the initial process by itself.
Appendix D: Single Type Component Generic Specification Model Example

```plaintext
-- channel declarations
datatype STATES =
  START. {0} | FINISH. {0} | NOTFINISHED. {0} | RESET. {0} | ERROR. {0} | IDLE. {0}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC1 = {{ chan0, chan1, chan2, chan3 }}
PROC1 = PROC_PROCESS_ANNOTATE_OUTER(chan0, chan1, chan2, chan3)
alpha_PROC4 = {{ chan0, chan1, chan2, chan3 }}
PROC4 = PROC_PROCESS_DESIRED GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC5 = PROC1

-- GenSpec1 with annotation events added by GenSpec 5 running in parallel
GEN_SPEC1_WITH_ANNOTATIONS = ( GEN_SPEC1 \ alpha_PROC1 \ GEN_SPEC5 )
-- The annotations only that were added to GenSpec1 by GenSpec 5
GEN_SPEC1_HIDDEN_ANNOTATIONS = (GEN_SPEC1_WITH_ANNOTATIONS \ {{ annotation \ }} )

-- Check that GenSpec4 with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC1_HIDDEN_ANNOTATIONS \FD. GEN_SPEC1
assert GEN_SPEC1 \FD. GEN_SPEC1_HIDDEN_ANNOTATIONS
```

Figure 70: Example of GenSpec Assertions 12 for Control Flow Process

D.2.13 GenSpec Assertions 13: GenSpec 4 With Signals Hidden Is Similar to GenSpec 6

These assertions (see Figure 71) demonstrates that GenSpec 4 correctly driven with the events that represent the low level signals hidden, performs in a similar manner to GenSpec 6. The processes can not be failures divergent checked ("\FD") both ways against each other, as the driving signal events (which are hidden) determine the initial annotation state that occurs. The hidden events become tau-e events preceding the initial annotations, which GenSpec 6 does not contain.
Appendix D: Single Type Component Generic Specification Model Example

```
-- channel declarations
datatype STATES = START. {0} | FINISH. {0} | NOTFINISHED. {0} | RESET. {0} | ERROR. {0} | IDLE. {0}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC0 = {[ chan0, chan1, chan2, chan3 ]}
PROC0 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)
alpha_PROC2 = {[ chan0, chan1, chan2, chan3 ]}
PROC2 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = ( PROC0 \ {internalChoice} )
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)

-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROL = ( GEN_SPEC4 [ alpha_PROC2 ] GEN_SPEC3 )
-- GenSpec4 limited by the control specification GenSpec3, annotation only
GEN_SPEC4_ANNOTATIONS_ONLY = (GEN_SPEC4_WITH_CONTROL \ alpha_PROC0)

-- Check that GenSpec4 with hidden signals is similar to GenSpec6
-- These can not be failures divergent checked `[FD]' both ways, as the low
-- level driving signals that precedes the corresponding annotation events and
-- determine their occurrences, are hidden.
assert GEN_SPEC4_ANNOTATIONS_ONLY [FD= GEN_SPEC6
assert GEN_SPEC6 [T= GEN_SPEC4_ANNOTATIONS_ONLY
```

Figure 71: Example of GenSpec Assertions 13 for Control Flow Process

D.3 Conclusion & Evaluation

The combination of the assertions covered in section D.2 links various properties of the various models covered in section D.1. This builds up confidence with the models specified so that implemented components can both utilise them in their own proofs and be checked against them. It provides processes so that a low hardware level model of the components can be refinement checked against, a higher clock cycle based conceptual specification to refinement check against, and a method to link the two types of models together to show consistency between them.
Appendix D: Single Type Component Generic Specification Model Example

D.4 Future Work

D.4.1 Linking Clock Cycle Annotations to a Higher Level Specification

The models that have been created stop at a clock cycle conceptual model of what is occurring, it is possible to link this model to a model that represents at a software level what is happening. This may be useful as deadlock in software becomes live lock in hardware, as it is impossible for digital clocked logic to be deadlocked (each logic component will always output a value each clock cycle).

If this is desired, the annotation specification (see section D.1.6) can be linked to a higher and simplified specification through selectively hiding specific events so long as the circuit is not reset. Resetting the circuit can be perceived as restarting the application, and as such, restarting the CSP model. The idle state is also not allowed to occur, as we are concerned with what this component does when it is triggered, as this would be when the code that this hardware represents would have been called.

The reason why this simplified model was not fully implemented into the compiler was that it was not deemed essential to demonstrate the validity of the concept of this work. It was also deemed that the benefits gained would not warrant the required time required to code up all the required segments.

D.4.1.1 GenSpec 7: Software Component Generic Spec Model

This model (see Figure 72) provides a software level based model of a generic control flow process. A process when started may or may not ever finish, and it can only be restarted again if it does finish.
Appendix D: Single Type Component Generic Specification Model Example

-- internal choice with STOP is provided as an alternative to the FINISH event,
-- this is because conceptually when started, a control flow process does not
-- have to finish and is dependent on itself or other internal components to
determine if it does or not.

PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(id) =
let
A =
   annotation.START.id ->
   ( STOP
      |-
      annotation.FINISH.id -> A
   )
within A

Figure 72: Example of Simplified Control Flow Annotation Specification

D.4.1.2 GenSpec Assertions 14: Linking GenSpec 6 to GenSpec 7

These assertions (see Figure 73) demonstrate that properties of GenSpec 6 are similar to GenSpec 7.

-- The annotation index value being used
annotation_id = 0

-- An instance of the annotation specification
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(annotation_id)

-- Checking the annotation specification while ensuring that reset does
-- not occur (running in parallel with STOP), and hiding clock cycle annotations
EXPECTED_SIMPLIFIED_ANNOTATION_SPEC =
   (GEN_SPEC6
      || { annotation.RESET "}
      STOP
   ) \ { annotation.x | x<-{ IDLE, NOTFINISHED } "}

-- The new simplified specification to check against, GenSpec7
SIMPLIFIED_ANNOTATION_SPEC = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(annotation_id)

-- Check that the two processes are equivalent
assert EXPECTED_SIMPLIFIED_ANNOTATION_SPEC [FD= SIMPLIFIED_ANNOTATION_SPEC
assert SIMPLIFIED_ANNOTATION_SPEC [T= EXPECTED_SIMPLIFIED_ANNOTATION_SPEC

Figure 73: Example of how to link Annotation Spec to Simplified Spec

D.4.2 Model Simplifications

Although the models are valid and correct, I have noted that they can be partially simplified. Through altering the design of the CSP process (see section D.1.3) that is used to ensure that components are driven correctly, it is possible to partially simplify the complexity of the required models (see Appendix H for example simplification) making them easier to mentally visualise. An example of this simplification is covered in
Appendix H, as due to time restrictions full integrations of this simplification into the compiler for all components was not possible.
Appendix E  Single Type Component

Implemented Model Example

This section will cover and explain the CSP models required for an implemented component, along with the assertions that need to be checked to link the models to each other, thus demonstrating that an implemented component performs as required and within the behaviours dictated by its generic super-type component (see Figure 74).
Appendix E: Single Type Component Implemented Model Example

Each individual component is modelled and run in parallel (may contain D.1.2)

Constrain input signals to disallow incorrect outer level driving

CSP Model of Net List (E.1.3)

Refinement (E.2.2)

Equivalent (E.2.5)

CSP Model of Net List disallowing incorrect driving (E.1.3||D.1.3)

Annotate outer layer

Correctly driven net-list with outer annotations ((E.1.3||D.1.3)||D.1.5)

Hide low level events and check annotations perform similarly to expected model (E.2.8)

Implementation Hardware Specification (E.1.1)

Deadlock Check (E.2.3)

Refinement (E.2.4)

Super Type Hardware Specification (D.1.1)

Implementation Clock Cycle Software Specification (E.1.4)

Hide subset of annotation events relating to clock cycle aspects

Implementation Software Specification (E.4.1.1)

Super Type Clock Cycle Software Specification (D.4.1.1)

Refinement (E.4.1.2)

Figure 74: Overview of proof structure for an implemented component

E.1 Models & Specifications

The 'internalChoice' event that may appear within the code examples has been utilised instead of internal choice (i.e. 'I-I') to enable 'chase' compression to applied if desired. The 'internalChoice' event must be hidden for the specifications to be valid, but if 'chase' compression has been chosen, the event should only be hidden after 'chase' has been applied, otherwise the specification becomes invalid.
Appendix E: Single Type Component Implemented Model Example

E.1.1 ImpSpec 1: Valid Low Level Behaviour

This CSP model (see Figure 75) which is similar to the model defined in section D.1.1, specifies all the valid and allowable low level behaviour that this implemented component may perform at its outer boundary. It may utilises internal choice to determine the possible output behaviour it can perform, although it is not a requirement (e.g. boolean true, boolean false, SKIP, STOP, all have well defined fixed behaviours that do not rely on other internal components). It is useful to note that some implemented components may have the allowable interface boundary behaviour that is identical to that of its generic super type component (e.g. boolean comparisons, PAR), where as other components will have an interface boundary behaviour that is a refinement of its super type component (e.g. boolean true, boolean false, SEQ).

```
PROC_PROCESS_IF_DESIRED_SPEC(clock, reset, start, finish) =
let
  A =
    start?x -> reset?y -> C(x, y) []
    reset?y -> start?x -> C(x, y)
  B =
    start?0 -> reset?y -> D(y) []
    reset?y -> start?0 -> D(y)
  C(x, y) =
    clock?1 -> finish!0 ->
      ( y == 1 & A []
        y == 0 &
          ( x == 0 & A []
            x == 1 & B )
      )
  D(y) =
    y == 1 & clock?1 -> finish!0 -> A []
    y == 0 & clock?1 ->
      ( internalChoice -> finish!1 -> A [] -- |--
        internalChoice -> finish!0 -> B )
within finish!0 -> A
```

Figure 75: Low Level 'IF' Component Desired Specification

E.1.2 ImpSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 76) is similar to the model covered in section D.1.2, the CSP model is the model covered in section E.1.1 but altered so that it will accept incorrectly driven input signals followed by explicitly defined deadlocking (i.e. STOP).
E.1.3 **ImpSpec 3: Model of Implemented Logic**

This CSP model (see Figure 77), is a model of the segment of logic that this component represents and is achieved through modelling the individual logic components and running them in parallel.
Appendix E: Single Type Component Implemented Model Example

-- Components used in the segment of logic being verified
alpha_PROC2 = \{ chan1, chan8, chan0, chan7 \}

PROC2 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan7, chan8, 3)

alpha_PROC7 = \{ chan9, chan4 \}

PROC7 = PROC_NOT(chan9, chan4)

alpha_PROC1 = \{ chan1, chan6, chan0, chan5 \}

PROC1 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan5, chan6, 2)

alpha_PROC8 = \{ chan6, chan8, chan10 \}

PROC8 = PROC_OR(chan10, chan6, chan8)

alpha_PROC9 = \{ chan9, chan3, chan7 \}

PROC9 = PROC_AND(chan7, chan9, chan3)

alpha_PROC10 = \{ chan4, chan3, chan5 \}

PROC10 = PROC_AND(chan5, chan3, chan4)

alpha_PROC0 = \{ chan4, chan1, chan3, chan0, chan2 \}

PROC0 = PROC_BOOLEANANNOTATED_SPEC(chan0, chan1, chan2, chan3, chan4, 1)

-- The outer level signals for the component being tested
SYSTEM_INTERFACE = \{ chan1, chan0, chan10, chan2 \}

-- The alphabet of signals used in the model of the logic
SYSTEM_ALPHA = \{ chan3, chan0, chan6, chan2, chan5, chan4, chan8, chan9, chan10, chan1, chan7 \}

-- System Declaration of the internal logic components and the signals they use
SYSTEM_LIST = < (PROC2, alpha_PROC2 ), (PROC7, alpha_PROC7 ),
PROC1, alpha_PROC1 ), (PROC8, alpha_PROC8 ),
PROC9, alpha_PROC9 ), (PROC10, alpha_PROC10 ),
PROC0, alpha_PROC0 )>

-- The logic model of the implemented component. 'ImpSpec3'
SYSTEM =

( REPL(SYSTEM_LIST) \ diff(SYSTEM_ALPHA, SYSTEM_INTERFACE)
 ) \ {internalChoice}

-- Used to run the logic components in parallel
REPL(p) =

let

INNER1(p1, a1, p2) = p1 \ [ inter(a1, a2) ] p2

INNER2( <(p1, a1)>^<(p2, a2)>^p3 ) =

null(p3) & INNER1(p1, a1, p2, a2)

not null(p3) & INNER2( <(INNER1(p1, a1, p2, a2), union(a1, a2)>^p3 )

INNER3( <(p1, _)> ) = p1

within ( null(p) & STOP

[]) length(p) = 1 & INNER3(p)

[] length(p) > 1 & INNER2(p) )

Figure 77: CSP Model of the Logic Circuit Segment of the Implemented 'IF' Component

A graphical depiction of the segment of logic circuit that this model represents can be seen in Figure 78.
E.1.4 ImpSpec 4: Annotation Only Specification

This CSP model (see Figure 79), is the expected behaviour of the 'IF' component from an annotation only perspective, with annotation models of the internal component (i.e. the boolean condition, the 'then' process and the 'else' process) having to be supplied. If the internal annotation components supplied are the corresponding generic specifications, the CSP model will demonstrate all the possible behaviours of the 'IF' component, describing both how driving the component drives the internal components and the internal components behaviour effects the outputs of this component. The reason why this model was designed to take in processes representing the internal components is so that if the supplied internal component specifications are a refinement of the corresponding generic specification, they will limit the behaviour dictated in the 'IF' component to that which describes what should conceptually occur in the hardware.

```
channel chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC : {0..2}
channel chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC : {0, 1}
channel chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC
PROC_PROCESS_IF_HIGHER_INNER_SPEC(id, bool, bid, thenproc, tid, elseproc, eid) =
let
  A =
    ( | \| x: {bool, thenproc, elseproc} o x |
    [ | { annotation.z.x,
       annotation.y.bid,
       annotation.w.v
    | v<-{tid, eid},
    xx<-{bid, tid, eid},
    y<-{BOOLEANREAD,
       BOOLEANREADALLOWED,
       BOOLEANTRUE,
       BOOLEANFALSE},
    z<-{RESET, IDLE},
    w<-{START, FINISH, NOTFINISHED}
    ]
    )
  B
-- combination of boolean and process
```
Appendix E: Single Type Component Implemented Model Example

```
B =
| C | {} annotation.RESET.x, |
|   | chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
|   | chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
|   | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC |
|   | x<-{bid, tid, eid, id} |
|   | |
| D |
| \ | {} chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
| | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
| | chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC |
|-- boolean part |
| C =
| let |
| CA = |
| | {} z:{bid, tid, eid, id} e annotation.RESET.z -> SKIP ); CA |
| | |
| | \ | {} z:{bid, id} e annotation.IDLE.z -> SKIP |
| | |
| | \ | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> CA |
| | ] |
| | (( annotation.BOOLEANREAD.bid -> SKIP ) |
| | |
| | ( annotation.START.id -> SKIP ) |
| | ) |
| | \ | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> |
| | ( CB |
| | \ | {} annotation.RESET.x, |
| | | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
| | | chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC, |
| | | chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC |
| | | x<-{id, bid, tid, eid} |
| | |
| | ) |
| | |
| | ) |
| | CB = |
| | annotation.BOOLEANNOTREAD.bid -> |
| | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> |
| | chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> |
| | ( {} z:{bid, tid, eid, id} e annotation.RESET.z -> SKIP ) |
| | |
| | \ | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 -> |
| | \ | chan_LINK_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 -> CC |
| | ) |
| | annotation.BOOLEANFALSE.bid -> |
| | annotation.BOOLEANTRUE.bid -> |
| | chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> |
| | chan_LINK_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> CC |
| | ) |
| | CC = |
| | {} z:{bid, tid, eid, id} e annotation.RESET.z -> SKIP |
| | |
| | \ | annotation.IDLE.bid -> chan_LINK_PROC_PROCESS_IF_HIGHER_INNER_SPEC |
```
Appendix E: Single Type Component Implemented Model Example

```plaintext
within CA
E =
  let
  EA =
    annotation.NOTFINISHED.id ->
      ( chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
        chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> EC
        []
      )
    ( chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 ->
      chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 ->
      chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -
      chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
      chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
      chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> EC
    )
  
  EB =
    ( ||| z:{bid, tid, eid, id} @ annotation.RESET.z -> SKIP )
    []
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
    ( chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
      annotation.NOTFINISHED.id ->
      chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> EB
      []
    )
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 ->
    annotation.FINISH.id ->
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> SKIP
  
  EC =
    ( ||| z:{bid, tid, eid, id} @ annotation.RESET.z -> SKIP )
    []
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> EA
  
within EA
D =
  let
  DA =
    ( ( ||| z:{bid, tid, eid, id} @ annotation.RESET.z -> SKIP ); DA )
    []
    ( ||| z:{tid, eid} @ annotation.IDLE.z -> SKIP )
    ;
    ( chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> DA
      []
    )
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> DB
  
  DB =
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
    ( ( ||| z:{bid, tid, eid, id} @ annotation.RESET.z -> SKIP ); DA )
    []
    ( ||| z:{tid, eid} @ annotation.IDLE.z -> SKIP )
    ;
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> DB
  
  []
  ( chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 ->
    chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> DF{tid, eid}
    []
  )
  chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 ->
```
Appendix E: Single Type Component Implemented Model Example

```plaintext
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> DF(eid, tid)

DA

DC(x) =
{ ||| z:[bid, tid, eid, id] @ annotation.RESET.z -> SKIP }

[ ]
annotation.START.x ->
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> DD(x)

DD(x) =
annotation.FINISH.x ->
chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 ->
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> SKIP

[ ]
annotation.NOTFINISHED.x ->
chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
( ( ||| z:[bid, tid, eid, id] @ annotation.RESET.z -> SKIP )

[ ]
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> DD(x)

DE(x) =
( ||| z:[bid, tid, eid, id] @ annotation.RESET.z -> SKIP )

[ ]
{ annotation.IDLE.x -> chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC ->
( chann_proc_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 ->
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> SKIP
[ ]
chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 ->
chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC -> DD(x)

)}

DF(x, y) =
{ DE(y)
  ||| ( chan_link_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
  chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
  annotation.RESET.z
  | z<-{id, bid, eid, tid}
  [ ]
  [ ]
  DC(x)
  )
within DA
within B

E.2 Assertions: Linking the Models Together

E.2.1 ImpSpec Assertion 1: Deadlock Freedom

The assertion stated in Figure 80 checks that the model of the segment of logic circuit is deadlock-free. This check demonstrates two properties, the first is that there exists no loops consisting of only clocked or non-clocked logic components (see section 4.4.2), the second is that any internal components are guaranteed to be driven correctly so long as the outer component is driven correctly. The guarantee that internal components are driven correctly is possible because the models of the internal components are models that
Appendix E: Single Type Component Implemented Model Example

accept all possible inputs (both valid and incorrect), with the incorrect inputs being followed by explicitly defined deadlock i.e. 'STOP' (see section D.1.2). If the STOP's are not reached, then invalid inputs to internal components have not been created or propagated through.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = { chan1, chan0, chan10, chan2 |
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = { chan1, chan0, chan10, chan2 |
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

-- Deadlock-free check the expected correct generic component
assert (IMP_SPEC3 || SYSTEM_INTERFACE || GEN_SPEC3 : [deadlock free [F]])
```

Figure 80: 'IF' Component Deadlock-Free Assertion

E.2.2 ImpSpec Assertion 2: Super Type Control Only Limits the Behaviour

The assertion stated in Figure 81 demonstrates that the process that dictates allowable correct driving input signals of the super type of this component, does only limit the behaviour of this implemented component, thus ensuring that it does not introduce any new behaviour.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = { chan1, chan0, chan10, chan2 |
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = { chan1, chan0, chan10, chan2 |
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

-- Check that the control specification only limits the behaviour of the segment
-- of logic, and does not introduce new behaviour
assert IMP_SPEC3 |- (IMP_SPEC3 || SYSTEM_INTERFACE || GEN_SPEC3)
```

Figure 81: Super Type Component Limits the Behaviour of the Implementation
Appendix E: Single Type Component Implemented Model Example

E.2.3 ImpSpec Assertion 3: Expected ‘IF’ Component Behaviour is Deadlock-Free

The assertion stated in Figure 82 demonstrates that the expected boundary behaviour that the model of the implementation of the logic circuit segment will be checked against is deadlock-free. This is to provide better confidence in this model's correctness as it will be used in future checks.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
assert IMP_SPEC1 : [deadlock free [F]]
```

Figure 82: Expected 'IF' Component Behaviour is Deadlock-Free

E.2.4 ImpSpec Assertion 4: Expected Boundary Behaviour Refines Super Type

The assertion stated in Figure 83 demonstrates that the expected correct boundary behaviour of the implemented component is a valid refinement of the expected boundary behaviour of its generic super type component.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
GEN_SPEC1 =
    ( PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan10)
    \ {internalChoice}
    )
assert GEN_SPEC1 \ T= IMP_SPEC1
```

Figure 83: Expected 'IF' Component Behaviour is a Refinement of Super Type
E.2.5 ImpSpec Assertions 5: Correctly Driven Implementation Behaves as Expected

The assertions stated in Figure 84 demonstrate that the segment of logic circuit for this implemented component, if driven correctly, behaves as expected.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = [{ chan1, chan0, chan10, chan2 | }
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = [{ chan1, chan0, chan10, chan2 | }
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

assert IMP_SPEC1 [FD=

( (IMP_SPEC3 [{ SYSTEM_INTERFACE | GEN_SPEC3} \ { | annotations | })
assert ((IMP_SPEC3 [{ SYSTEM_INTERFACE | GEN_SPEC3}
\ { | annotations | })

) [FD= IMP_SPEC1
```

Figure 84: Correctly Driven 'IF' Component Behaves as Expected

E.2.6 ImpSpec Assertion 6: Annotation Outer Level Does Not Introduce Deadlock

The assertion stated in Figure 85 demonstrates that annotating the outer level of the implemented component with the annotation process specified by its super type (i.e. GenSpec 5, see Section D.1.5), does not introduce any deadlocks.
Appendix E: Single Type Component Implemented Model Example

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
-- Create an instance of the models to check
-- The alpha PROC contains the low level channels used by the processes
alpha_PROC6 = { [ chan1, chan0, chan10, chan2 ] }
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

alpha_PROCS = { [ chan1, chan0, chan10, chan2 ] }
PROC5 = PROC_PROCESS_ANNOTATE_OUTER(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = { [ chan1, chan0, chan10, chan2 ] }
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6
GEN_SPEC5 = PROC5

assert ( (IMP_SPEC3 [] SYSTEM_INTERFACE [] GEN_SPEC3)

[ [] SYSTEM_INTERFACE []] GEN_SPEC5
) : (deadlock free [F])
```

Figure 85: Annotating outer level of 'IF' component does not introduce deadlock

E.2.7 ImpSpec Assertion 7: Expected High Level Behaviour is Deadlock-Free

The assertion stated in Figure 86 demonstrates that the high level model describing the expected behaviour of the implemented component is deadlock-free. This helps to build confidence in the model for when using it in future checks.

```plaintext
-- Create an instance of the models to check
IMP_SPEC4 = PROC_PROCESS_IF_HIGHER_INNER_SPEC(0,
HIGHER_SPEC0_1, 1,
HIGHER_SPEC1_2, 2,
HIGHER_SPEC2_3, 3)

-- Higher Process Instances
HIGHER_SPEC1_2 = PROC_PROCESS_HIGHER_OUTER_SPEC(2)
HIGHER_SPEC2_3 = PROC_PROCESS_HIGHER_OUTER_SPEC(3)
HIGHER_SPEC0_1 = PROC_BOOLEAN_HIGHER_OUTER_SPEC(1)

assert IMP_SPEC4 : (deadlock free [F])
```

Figure 86: 'IF' Component High Level Behaviour is Deadlock-Free
E.2.8 ImpSpec Assertions 8: Component Behaves Similarly to Expected Higher Spec

The assertions stated in Figure 87 demonstrate that the annotations obtained from the implemented segment of logic circuit, performs in a similar manner to that of the expected higher behavioural specification. The test can not be failure divergence checked both ways (one has to be a trace refinement), this is due to the way annotations are added to the outer layer. As the outer level input annotations occur after the corresponding low level input signal events (i.e. the events that represent the wires), hiding these low level signal events causes the high level model extracted from the implemented segment of logic circuit to appear to have internal choice determining the high level conceptual input states. The internal choice for the inputs does not really exist, but appears because the events that do determine what occurs though external choice have been hidden (i.e. the low level signals). Through altering the process of annotating the outer level of a component (see Appendix H), it is possible to simplify the extracted model so that it directly equivalent to the expected higher behaviour.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = {{chan1, chan0, chan10, chan2}}
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

alpha_PROC5 = {{chan1, chan0, chan10, chan2}}
PROC5 = PROC_PROCESS_ANNOTATE_OUTER(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = {{chan1, chan0, chan10, chan2}}
IMP_SPEC3 = PROC6
GEN_SPEC3 = PROC5

assert (( (IMP_SPEC3 [[SYSTEM_INTERFACE ]] GEN_SPEC3)
          [[SYSTEM_INTERFACE ]] GEN_SPEC5
          \ SYSTEM_INTERFACE
        )[FD=Imp_Model_3
assert Imp_Model_3[T= ((IMP_SPEC3 [[SYSTEM_INTERFACE ]] GEN_SPEC3)
          [[SYSTEM_INTERFACE ]] GEN_SPEC5
          \ SYSTEM_INTERFACE )
```

Figure 87: 'IF' Component Behaves Similarly to Expected Higher Behaviour
E.3 Conclusions & Evaluation

The combination of the assertions covered in section E.2 links various properties of the various models covered in section E.1 together and also with some of the models covered in section D.1 (i.e. the models for its corresponding super type). This builds up confidence with the implemented component through crosschecking various properties hold true throughout the various models created for it, along with the implementation being a refinement of its super type component, thus allowing the implemented component to be placed wherever its super type has been used.

E.4 Future Work

E.4.1 Linking Clock Cycle Annotations to Higher Specification

Similar to the work described in section D.4.1, the CSP model covered in section E.1.4 could be linked to a higher conceptual description of the component that gives the sequencing of the required conceptual events at a software level, and not a hardware clock cycle level.

E.4.1.1 ImpSpec 5: Software Specification Model

This model (see Figure 88) provides a software level based model of the implemented component. The external choice with ‘STOP’ is to provide a clear indication of where the internal components behaviour is expected to create possible deadlocking within this model. The deadlocking at within this model is allowed to possibly occur under the conditions where when an internal component is started, it never completes. Should this condition arise, the ‘IF’ component will never finish, a simple example of this is if the ‘then’ component gets triggered and is a ‘while(true)’ loop, the ‘while(true)’ loop never finishes, and so the ‘IF’ component would never finish.
Appendix E: Single Type Component Implemented Model Example

PROC_PROCESS_IF HIGHER SIMPLIFIED_SPEC(id, bool, bid, thenproc, tid, elseproc, eid) =

let
    A =
        { annotation.START.id -> SKIP
        annotation.BOOLEANREAD.bid -> SKIP
        }
    ( STOP
    []
    annotation.BOOLEANREADALLOWED.bid ->
        { annotation.BOOLEANTRUE.bid -> B(tid)
            []
            annotation.BOOLEANFALSE.bid -> B(eid)
        }
    )
    B(x) =
        annotation.START.x ->
            { STOP
            []
                annotation.FINISH.x ->A
            }
    C =
        { [ ]
            x: {bool, thenproc, elseproc} @ x
            [ ]
                [ ]
                    annotation.BOOLEANREAD.bid,
                    annotation.BOOLEANREADALLOWED.bid,
                    annotation.BOOLEANTRUE.bid,
                    annotation.BOOLEANFALSE.bid,
                    annotation.START.y,
                    annotation.FINISH.y,
                    | y <- {tid, eid}
            []
        }
    A
    )
within C

Figure 88: 'IF' Component Software Annotation Behavioural Specification

E.4.1.2 ImpSpec Assertion 9: Software Specification is a Refinement of Super Type

The assertion stated in Figure 89 demonstrates that the expected higher software specification for the implemented component with its internal events hidden is a refinement of its super types' software specification model. The example show happens to be equivalent to its super type software specification model, but this is not a requirement and is why it is not being tested for.
Appendix E: Single Type Component Implemented Model Example

```
-- Generic Boolean Higher Software Specification
PROC_BOOLEAN_HIGHER_SIMPLIFIED_SPEC(id) =
   let
      A =
         annotation.BOOLEANREADALLOWED.id ->
            ( STOP
            | annotation.BOOLEANREADALLOWED.id ->
               ( annotation.BOOLEANTRUE.id -> A
               | annotation.BOOLEANFALSE.id -> A
            )
      )
   Within A

-- Internal Components
BoolTest = PROC_BOOLEAN_HIGHER_SIMPLIFIED_SPEC(1)
ThenProc = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(2)
ElseProc = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(3)

-- Components to Test
IMP_SPECS5 = PROC_PROCESS_IF_HIGHER_SIMPLIFIED_SPEC(0, BoolTest, 1, ThenProc, 2, ElseProc, 3)

GEN_SPEC7 = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(0)
IMP_SPECS5_HIDDEN_INTERNALS =
   \ diff( { | annotations | },
            { annotations.x.0 | x <- { START, FINISH } } )

-- Check that ImpoSpec5 with internal events hidden is a refinement of its super type
assert GEN_SPEC7 [T= IMP_SPECS5_HIDDEN_INTERNALS

Figure 89: Higher Software Specification is a Refinement of the Super Type
```
Appendix F  Multi-Type Component
Generic Specification Model Example

This section will cover and explain the CSP models required for a multi-type generic super-type component, along with the assertions that need to be checked to link the models to each other and to the corresponding single-type generic super-type components for the interfaces.

F.1  Models & Specifications

The 'internalChoice' event that may appear within the code examples has been utilised instead of internal choice (i.e. ‘|–|’) to enable 'chase' compression to applied if desired. The 'internalChoice' event must be hidden for the specifications to be valid, but if 'chase' compression has been chosen, the event should only be hidden after 'chase' has been applied, otherwise the specification becomes invalid.

F.1.1  GenSpec 1: Valid Low Level Behaviour

This model specifies all the valid and allowable low level behaviour of this type of multi-type super type component. The purpose is to describe the interface boundary behaviours, thus enabling implemented components to refinement check against it proving there behaviours are within the requirements for it to be a sub-type of this super-type.

This specification (see Figure 90) will only accept correct input driving signals, and will return valid output result signals. Internal choice is utilised to enable it to specify all the possible valid refinements.

Figure 90: Low Level Generic Data Storage Specification

| channel chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC: {0..2} |
| channel chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC |
| channel chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC: {0..4} |
| channel chan_readbits_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC : {0..2}.{0, 1} |
| channel chan_storebits_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC: {0..2}.{0, 1} |

PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC(clock, reset, stores, reads) =
let
  A =
Appendix F: Multi-Type Component Generic Specification Model Example

\[
\text{length(stores)} = 0 \text{ and length(reads)} = 0 \& \ E \\
\text{length(stores)} = 0 \text{ and length(reads)} = 0 \& \ B \\
\text{length(stores)} = 0 \text{ and length(reads)} = 0 \& \ C \\
\text{length(stores)} = 0 \text{ and length(reads)} = 0 \& \ D
\]

\[B = \]

\[
\begin{align*}
\text{BA =} \\
& \{\{\} \text{ union} \{ \\
& \text{reset,} \\
& \text{clock,} \\
& \text{chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC} \\
& \text{chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.y} \\
& \text{y}\in\{3,4\} \\
& \}\} \\
& \}\xrightarrow{x:\text{set(stores)} \& \text{BB(x)}} \\
& \}\xrightarrow{x:\text{reset.}1,} \\
& \text{clock,} \\
& \text{chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC,} \\
& \text{chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC} \\
& \}\xrightarrow{x:\text{set(stores)} \& \text{BBB(x)}} \\
& \}\xrightarrow{\text{BBB (store, stored, data) } =} \\
& \text{let} \\
& \text{BBA =} \\
& \{\text{BBB} \\
& \{\{\} \text{ clock,} \\
& \text{chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC,} \\
& \text{chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC} \\
& \}\} \\
& \}\xrightarrow{\text{BBB (store, stored, data) } =} \\
& \text{let} \\
& \text{BBB =} \\
& \{\text{chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC} \} \\
& \}\xrightarrow{x:\text{set(data)} \& \text{BBB}} \\
& \text{chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0} \rightarrow \\
& \{\text{ reset.}0 \rightarrow \text{SKIP} \} \\
& \text{chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.1} \rightarrow \\
& \{\text{ set(data)} \& \text{x}\rightarrow \text{SKIP} \} \\
& \text{chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.1} \rightarrow \\
& \{\text{ store}0 \rightarrow \text{BBB} \} \\
& \}\xrightarrow{\text{reset} \rightarrow} \\
& \text{chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC} \rightarrow \text{BBB}
\end{align*}
\]
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
SKIP

; BBD

reset?0 ->
( store?0 ->
  chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0 -> BBD
[]
  store?1 ->
  chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.1 ->
  chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0 --> BBE
)
[]

store?0 ->
chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0 ->
( ( reset?0 -> SKIP
  []
    reset?1 -> SKIP
  ;
  BBD
 )
[]

store?1 ->
chan_mid_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.1 ->
( reset?0 ->
  chan_PROC STORAGECOMPONENT_DESIRED GENERIC SPEC.0 --> BBE
[]
  reset?1 -- BBD
 )

BBD =

clock?1 ->
( ( chan_PROC STORAGECOMPONENT_DESIRED GENERIC Spec.3 --> SKIP
  []
    chan_PROC STORAGECOMPONENT_DESIRED GENERIC Spec.4 --> SKIP
  ;
  ( stored?0 ->
    chan_link_PROC STORAGECOMPONENT_DESIRED GENERIC Spec. ->
    BBC
  )
 )

BBF =

clock?1 ->
( ( chan_PROC STORAGECOMPONENT_DESIRED GENERIC Spec.3 -->
  stored?1 --
    chan_link_PROC STORAGECOMPONENT_DESIRED GENERIC Spec. -->
    BBC
  []
  chan_PROC STORAGECOMPONENT_DESIRED GENERIC SPEC.4 -->
  stored?0 ->
  chan_link_PROC STORAGECOMPONENT_DESIRED GENERIC SPEC. -->
BBF
 )

BBF =

reset?1 --> store?0 ->
  chan_mid_PROC STORAGECOMPONENT_DESIRED GENERIC_SPEC.0 --> BBD
[]

reset?0 --> store?0 ->
  chan_mid_PROC STORAGECOMPONENT_DESIRED GENERIC_SPEC.0 --> BBG
[]

store?0 ->
chan_mid_PROC STORAGECOMPONENT_DESIRED GENERIC SPEC.0 ->
( reset?1 --> BBD
[]
  reset?0 --> BBG
 )

BBG =
```

147
Appendix F: Multi-Type Component Generic Specification Model Example

clock?1 -> chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4
storedl0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC -> BBF
within ( storedl0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC -> BBA
)
within BA
C =
let
CA =
( ( [ ] union{
  [ reset,
    clock,
    chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC
  ])
, chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.y
  y<-[3,4] }
  )
) x: set(reads) © CB(x)
[[ [ [ reset.1,
    clock,
    chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC,
    chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC
  ])
  ]
( chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC -> F )
) \ [ chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC,
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ]

CB( (read, readallowed, data) ) =
let
CBA =
reset?1 ->
  ( ( read?0 -> SKIP
     []
     read?1 -> SKIP
   )
   , CBB
 )
reset?0 ->
  ( read?0 -> CBB
     []
     read?1 -> CBC
   )
[]
read?0 ->
  ( ( reset?0 -> SKIP
     []
     reset?1 -> SKIP
   )
   , CBB
 )
[]
read?1 ->
  ( reset?0 -> CBC
     []
     reset?1 -> CBB
   )
CBB =
clock?1 ->
  ( ( chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 -> SKIP
     []
     chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 -> SKIP
   )
148
Appendix F: Multi-Type Component Generic Specification Model Example

```
chan_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC.4 -> SKIP

( readallowed!0 ->
  ( x: set(data) @ x!0 -> SKIP )
)

(chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC ->
  CBA
)

CBC =

  clock?1 ->
  ( chan_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC.3 ->
    readallowed!1 ->
    ( x: set(data) @ x!0 -> SKIP )
    ;
    ( chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC ->
      CBA
    )
    )
  )
)

[]

chan_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC.4 -> readallowed!0 ->

( x: set(data) @ x!0 -> SKIP )

( chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC ->
  CBD
)

)

CBD =

  reset?1 -> read?0 -> CBB
[
]

reset?0 -> read?0 -> CBE
[
]

read?0 ->

  ( reset?1 -> CBB
[
]
  reset?0 -> CBE
)

CBE =

  clock?1 -> chan_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC.4 -> readallowed!0 ->

  ( x: set(data) @ x!0 -> SKIP )

  ( chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC ->
    CBD
  )
)

within ( readallowed!0 ->

  ( x: set(data) @ x!0 -> SKIP )

  ( chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC ->
    CBA
  )
)

within CA

D =

let

DA =

  ( ( ( ) union

    ( [ reset,
      clock,
      chan_link_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC
    ]
    )

  ) chan_PROC_STORAGECOMPONENT_DESIRED GENERICSPEC )

```
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
// Example of a multi-type component generic specification

let DBA = DBB.[union(] reset, clock, chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC [)]
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC. y = [3, 4]

let DBA = DBB.[union(] reset, clock, chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC [)]
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC. y = [3, 4]

let DB = (store, stored, data)
let DBB = (chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC, chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC, clock)
let DBB = (chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC)
let DBA = (chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC, chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC, clock)
let DB = (store, stored, data)
let DBA = (chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC)
let DBB = (chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC)
```

150
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.1 ->
  ( DBBB(data)
  
  DBBA
  )
DBBB(x) =
  length(x) == 0 & STOP
  
  length(x) == 1 & DBBC(head(x), length(data) - 1)
  
  length(x) > 1 &
  ( DBBB(head(x), length(data) - length(x))
    | chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC,
    | chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.2,
    | clock.1
    |
    | DBBB(tail(x))
  )
DBBC(x, y) =
  ( ( x?0 ->
    
    chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.2 ->
    chan_storebits_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.y.0 ->
    clock?1 -> SKIP
    
    clock?1 -> SKIP
    
    x?1 ->
    
    chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.2 ->
    chan_storebits_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.y.1 ->
    clock?1 -> SKIP
  )
  )
  
  ( chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
    SKIP
  )
  
  within DBBA
DBC =
  reset?1 ->
  ( ( store?0 ->
      chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0 ->
      SKIP
    
    store?1 ->
    chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.1 ->
    SKIP
    
    ;
    DBD
  )
  )
  
  reset?0 ->
  ( store?0 ->
    chan_mid_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.0 ->
    DBD
    
    store?1 ->
    chan_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.0 ->
    chan_mid_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.1 ->
    chan_mid_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.2 ->
    DBE
```
Appendix F: Multi-Type Component Generic Specification Model Example

\[
\begin{align*}
\text{store? } 0 & \rightarrow \\
\quad \text{chan } \_\text{mid} & \text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 0 \\
& \rightarrow \\
& \quad ( \text{reset? } 0 \rightarrow \text{SKIP} \\
& \quad [ ] \\
& \quad \text{reset? } 1 \rightarrow \text{SKIP} \\
& \quad ; \\
& \quad \text{DBD} \\
& \quad [ ] \\
\text{store? } 1 & \rightarrow \\
\quad \text{chan } \_\text{mid} & \text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 1 \\
& \rightarrow \\
& \quad ( \text{reset? } 0 \rightarrow \\
& \quad \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 0 \\
& \quad \rightarrow \\
& \quad \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 2 \rightarrow \\
& \quad \text{DBE} \\
& \quad [ ] \\
& \quad \text{reset? } 1 \rightarrow \text{DBD} \\
& \quad ) \\
\text{DBD} = \\
\quad \text{clock? } 1 & \rightarrow \\
& \quad ( \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 3 \rightarrow \text{SKIP} \\
& \quad [ ] \\
& \quad \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 4 \rightarrow \text{SKIP} \\
& \quad ; \\
& \quad ( \text{stored? } 0 \rightarrow \\
& \quad \text{chan } \_\text{link } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC} \rightarrow \\
& \quad \text{DBC} \\
& \quad ) \\
\text{DBE} = \\
\quad \text{clock? } 1 & \rightarrow \\
& \quad ( \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 3 \rightarrow \\
& \quad \text{stored? } 0 \rightarrow \\
& \quad \text{chan } \_\text{link } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC} \rightarrow \\
& \quad \text{DBC} \\
& \quad [ ] \\
& \quad \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 4 \rightarrow \\
& \quad \text{stored? } 0 \rightarrow \\
& \quad \text{chan } \_\text{link } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC} \rightarrow \\
& \quad \text{DBF} \\
\text{DBF} = \\
\quad \text{reset? } 1 & \rightarrow \text{store? } 0 \rightarrow \\
& \quad \text{chan } \_\text{mid } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 0 \rightarrow \text{DBD} \\
& \quad [ ] \\
\text{reset? } 0 & \rightarrow \text{store? } 0 \rightarrow \\
& \quad \text{chan } \_\text{mid } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 0 \rightarrow \text{DBG} \\
& \quad [ ] \\
\text{store? } 0 & \rightarrow \\
& \quad \text{chan } \_\text{mid } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 0 \rightarrow \\
& \quad ( \text{reset? } 1 \rightarrow \text{DBD} \\
& \quad [ ] \\
& \quad \text{reset? } 0 \rightarrow \text{DBG} \\
& \quad ) \\
\text{DBG} = \\
\quad \text{clock? } 1 & \rightarrow \\
& \quad \text{chan } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC. } 4 \rightarrow \\
& \quad \text{stored? } 0 \rightarrow \\
& \quad \text{chan } \_\text{link } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC} \rightarrow \\
& \quad \text{DBF} \\
\quad \text{within stored? } 0 \rightarrow \\
& \quad \text{chan } \_\text{link } \_\text{PROC STORAGECOMPONENT DESIRED GENERIC SPEC} \rightarrow \\
& \quad \text{DBA} \\
\text{DC} \left( \text{read, readallowed, data} \right) =
\end{align*}
\]

152
let
DCA =
   reset?0 ->
      ( [ read?0 -> SKIP
           read?1 -> SKIP
         ] ;
       DCB
     )
//
reset?0 ->
   ( [ read?0 -> DCB
     ]
     read?1 ->
       chan_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC.1 ->
       DCC
   )
//
reset?0 ->
   ( [ reset?0 -> SKIP
       ]
     reset?1 -> SKIP
   )
//
DCB
//
reset?1 ->
   ( reset?0 ->
      chan_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC.1 ->
      DCC
   )
//
reset?1 -> DCB
//
DCB =
clock?1 ->
   ( [ chan_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC.3 -> SKIP
       chan_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC.4 -> SKIP
     ]
     readallowed:0 ->
      ( [ ||| x:set(data) @ x:0 -> SKIP ] )
     ;
     ( chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC -> DCA )
   )
//
DCC =
   ( [ DCD
        [ || { chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC,
               readallowed,
               chan_PROC_STORAGECOMPONENT_DESIRED GENERIC SPEC,
               reset,
               read,
               clock
             ]
        ]
      ]
   )
//
DCE
//
DCD =
   let
   DCDA =
clock?1 ->
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
( chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
  DCDB
[]
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
  SKIP
[]
reset?1 -> read?0 -> clock?1 ->
  chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
  SKIP
[]
reset?0 -> read?0 -> clock?1 ->
  chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC ->
  DCDB
[]
read?0 ->
  ( reset?1 -> clock?1 ->
    chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC ->
    SKIP
[]
reset?0 -> clock?1 ->
  chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC ->
  DCDB
]
within DCDA
DCEA =
  let
  DCEA(x) =
    length(x) == 0 & STOP
[]
    length(x) == 1 &
      chan_readbits_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.( length(data) - length(x)
      )?z ->
      DCEB(head(x), z)
[]
    length(x) > 1 &
      ( chan_readbits_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.( length(data) - length(x)
      )?z ->
      DCEB(head(x), z)
    )
[]
chan_link_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC,
readallowed,
chan_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC,
reset,
read,

  clock
    .
  )
[]
DCEB(x, y) =
  chan_PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC.4 ->
```

154
Appendix F: Multi-Type Component Generic Specification Model Example

readallowedl0 ->
x!0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
DCEC(x)
[

chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
readallowedl1 ->
x!y ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
SKIP
]

DCEC(x) =
reset?1 -> read?0 -> clock?1 ->
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
readallowedl0 ->
x!0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
SKIP
[

reset?0 -> read?0 -> clock?1 ->
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4 ->
readallowedl0 ->
x!0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
DCEC(x)
[

read?0 ->
 ( reset?1 -> clock?1 ->
 chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
 readallowedl0 ->
x!0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
SKIP
[

reset?0 -> clock?1 ->
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.4 ->
readallowedl0 ->
x!0 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
DCEC(x)
]

within clock?1 -> DCEA(data)
within ( readallowedl0 ->
 ( ( ||| x:set(data) @ x!0 -> SKIP )
 ;
 ( chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
 DCA
 )
 )
)

within DA
-- No Reads or Stores
E =
 ( reset?0 -> SKIP
 []
 reset?1 -> SKIP
 ); ( clock?1 -> E )
F =
let
 FA =
 reset?1 -> clock?1 ->
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
FA
[

]clock?1 ->
chan_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC.3 ->
chan_link_PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC ->
FA
Appendix F: Multi-Type Component Generic Specification Model Example
F.1.2 GenSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 91) is based on the one covered in section F.1.1, but with the altered fact that it also accepts invalid driving input signals to be submitted to it. These invalid input driving signals are followed by an explicitly defined 'STOP', that will explicitly deadlock the model should it ever be reached. Similar to the specification in section F.1.1, the returned output signals will be all possible valid permutations allowed (internal choice is utilised to create those permutations, so long as it is driven correctly).

The reason why this model will accept invalid driving signals is to enable the model of any component connected to it the opportunity to provide any driving signals it may choose, this process will not limit or remove the possibility for the other component models to provide invalid signals to this one as an option when they are run in alphabetised parallel. The purpose of this is to enable possibility to check that if this specification is used as an internal component, so long as the outer component is driven correctly, this component will be driven correctly.

Figure 91: Low Level Generic Data Storage Specification with Explicit Deadlocking

```
channel chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC: 0..2
channel chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
channel chan_PROC_STORAGECOMPONENT_GENERIC_SPEC: 0..4
channel chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC: 0..2.\{0, 1\}
channel chan_storebits_PROC_STORAGECOMPONENT_GENERIC_SPEC: 0..2.\{0, 1\}
PROC_STORAGECOMPONENT_GENERIC_SPEC(clock, reset, stores, reads) =
let
  A =
    length(stores) == 0 and length(reads) == 0 & E
    | length(stores) != 0 and length(reads) == 0 & B
    | length(stores) == 0 and length(reads) != 0 & C
    | length(stores) != 0 and length(reads) != 0 & D
  B =
    let
      BA =
        ( { | | union{
            | | reset,
            | | clock,
            | | chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
            | |
            | chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.y
            | | y<-\{3,4\}
          |
        | x:set(stores) & BB(x)
      | | {
        | | reset.1,
        | | clock,
        | | chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC,
        | | chan_PROC_STORAGECOMPONENT_GENERIC_SPEC
      | |
    }
```
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
(module (chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC -> F)
   \ { chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC,
   chan_PROC_STORAGECOMPONENT GENERIC_SPEC
   } { (store, stored, data) } =
   let
   BBA =
     ( BBS
       { clock,
       chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC,
       chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC
       } )
   BBS =
     ( chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.0 ->
       ( x: set(data) @
         [ x?0 -> SKIP
           []
         ]
       x?1 -> STOP
       )
     )
     chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.1 ->
     ( x: set(data) @
       [ x?0 -> SKIP
       []
       ]
       x?1 -> SKIP
     )
     clock?1 ->
     chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC ->
     BBS
   )
   BBC =
   reset?1 ->
   ( store?0 ->
     chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.0 ->
     SKIP
     []
   store?1 ->
     chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.1 ->
     SKIP
   )
   BBD
   )
   reset?0 ->
   ( store?0 ->
     chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.0 -> BBD
     []
   store?1 ->
     chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.1 ->
     chan_PROC_STORAGECOMPONENT GENERIC_SPEC.0 ->
     BBE
   )
   )
   store?0 -> chan_mid_PROC_STORAGECOMPONENT GENERIC_SPEC.0 ->
   ( ( reset?0 -> SKIP
     []
   reset?1 -> SKIP
```

158
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext

let

BBD

; BBD

{}

store? 1 -> chan_midPROC_STORAGECOMPONENT_GENERIC_SPEC.1 ->

{ reset? 0 -> chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 -> BBE

{}reset? 1 -> BBD

BBD =

clock? 1 ->

{ chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 -> SKIP

{}chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 -> SKIP

{}stored! 0 -> chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> BBC

BBE =

clock? 1 ->

{ chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->

{}stored! 1 -> chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> BBC

{}chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->

{}stored! 0 -> chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> BBF

BBF =

reset? 1 ->

{ store? 0 ->

{}chan_midPROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->

{}BBD

{}store? 1 -> STOP

{}reset? 0 ->

{ store? 0 ->

{}chan_midPROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->

{}BBG

{}store? 1 -> STOP

{}reset? 0 ->

{ store? 0 -> chan_midPROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->

{}reset? 1 -> BBD

{}reset? 0 -> BBG

{}store? 1 -> STOP

{}reset? 0 ->

{}chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->

{}stored! 0 ->

{}chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> BBF

within stored! 0 -> chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> BBA

C =

let


159
```
Appendix F: Multi-Type Component Generic Specification Model Example

CA =
( ( [] union
( [] reset,
  clock,
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
 ])
  [ chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.y
    y=\{3,4\} ]
)
[] x: set(reads) & CB(x)
[] [] reset.l,
  clock,
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC,
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC
[]
[] chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> F
)
[] chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
chan_PROC_STORAGECOMPONENT_GENERIC_SPEC
CB( (read, readallowed, data) ) =
let
  CBA =
  reset?1 ->
( ( read?0 -> SKIP
  []
  read?1 -> SKIP
 )
; CBB
)
[]
reset?0 ->
( read?0 -> CBB
[]
  read?1 -> CBC
 )
[]
read?0 ->
( ( reset?0 -> SKIP
  []
  reset?1 -> SKIP
 )
; CBB
)
[]
read?1 ->
( reset?0 -> CBC
[]
  reset?1 -> CBB
 )
CBB =
  clock?1 ->
( ( chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 -> SKIP
  []
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 -> SKIP
 )
; (readallowed?0 ->
  ( [] x: set(data) & x?0 -> SKIP )
 )
; ( chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> CBA )

160
Appendix F: Multi-Type Component Generic Specification Model Example

CBC =
  clock? 1 ->
  ( chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 -> readallowed!1 ->
    ( [ ]
      x: set(data) @ x!0 -> SKIP
    )
    ;
    ( chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC -> CBA )
  )
[ ]
chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 -> readallowed!0 ->
  ( [ ]
    x: set(data) @ x!0 -> SKIP
  )
  ;
  ( chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> CBD )
)

CBD =
  reset? 1 ->
  ( read?0 -> CBB
    [ ]
    read?1 -> STOP
  )
[ ]
reset? 0 ->
  ( read?0 -> CBE
    [ ]
    read?1 -> STOP
  )
[ ]
read? 1 ->
  ( reset?1 -> CBB
    [ ]
    reset?0 -> CBE
  )
[ ]
read? 0 ->
  ( reset?1 -> CBB
    [ ]
    reset?0 -> CBE
  )
[ ]
read? 1 -> STOP

CBE =
  clock? 1 ->
  chan_PROC_STORAGECOMPONENT GENERIC_SPEC.4 ->
  readallowed!0 ->
  ( [ ]
    x: set(data) @ x!0 -> SKIP
  )
  ;
  ( chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC -> CBD )
)
within ( readallowed!0 ->
  ( [ ]
    x: set(data) @ x!0 -> SKIP
  )
  ;
  ( chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC -> CBA )
)

within CA

D =
  let
  DA =
    ( ( [ ]
        union(
          [ [ ]
            reset,
            clock,
            chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC
          [ ]
        ],
        [ chan_PROC_STORAGECOMPONENT GENERIC_SPEC.y
          | y<>{3,4} ]
      )
      | x: set(stores) @ DB(x)
    )
  union(
    [ [ ]
      reset,
      clock,
    ]
  )

161
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
|
| chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.y
| y<-{3,4}
|

union

reset.

Clock

chap link PROC STORAGECOMPONENT GENERIC SPEC
|
| chan_PROC_STORAGECOMPONENT_GENERIC_SPEC
|
| chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.y
| y<-{3,4}
|

x:set(reads) @ DC(x)
|

reset.

chan_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_storebits_PROC_STORAGECOMPONENT_GENERIC_SPEC,
clock,
chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC
|

( chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> H
|
| chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_storebits_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_PROC_STORAGECOMPONENT_GENERIC_SPEC
|

DB( (store, stored, data) ) =
let
DBA =
( DBB
|
| chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC,
chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC,
clock
|

DBC
|
| chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC
|

DBB =
let
DBBA =
( chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
| x:set(data) @
| x?0 -> SKIP
| []
| x?1 -> STOP
|

)|

clock?1 ->
chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
DBBA
|

|
}
chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 ->
( DBBB[data]
|

162
```
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
store? 0 -> chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
  ( ( reset? 0 -> SKIP
    []
    reset? 1 -> SKIP
  )
  DBD
)

store? 1 -> chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 ->
  ( reset? 0 ->
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
    chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.2 ->
    DBE
  []
  reset? 1 -> DBD
)

DBD =
  clock? 1 ->
  ( ( chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 -> SKIP
    []
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 -> SKIP
  )
  ;
  ( stored? 0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    DBC
  )
)

DBE =
  clock? 1 ->
  ( chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
    stored? 1 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    DBC
  []
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->
  stored? 0 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  DBF
)

DBF =
  reset? 1 ->
  ( store? 0 ->
    chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
    DBG
  []
  store? 1 -> STOP
)

reset? 0 ->
  ( store? 0 ->
    chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
    DBG
  []
  store? 1 -> STOP
)

store? 0 -> chan_mid_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 ->
  ( reset? 1 -> DBD
  []
  reset? 0 -> DBG
)

store? 1 -> STOP
DBG =
  clock? 1 ->
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->
```

164
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
storedl0 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> DBF
with ( storedl0 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  DBA )
DC( (read, readallowed, data) ) =
  let
    DCA =
      reset?1 ->
      ( (read?0 -> SKIP
        [ ]
        read?1 -> SKIP
      )
      DCB
    )
    [ ]
    reset?0 ->
    ( read?0 -> DCB
    [ ]
    read?1 -> chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 -> DCC
    )
    [ ]
    read?0 ->
    ( (reset?0 -> SKIP
    [ ]
    reset?1 -> SKIP
    )
    DCB
    )
    [ ]
    reset?1 ->
    ( reset?0 -> chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 -> DCC
    [ ]
    reset?1 -> DCC
    )
  DCB =
  clock?1 ->
  ( (chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 -> SKIP
  [ ])
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 -> SKIP
  )
  ;
  (readallowed?0 ->
    ( ||| x: x=set(data) @ x:0 -> SKIP )
  )
  ;
  ( chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> DCA )
  )
DCC =
  ( (DCE
  [ ]
  { chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC,
  readallowed,
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC,
  reset,
  read,
  clock
  }[
  ]
  DCE
  )
  ;
  DCA
  )
DCD =
```

165
let DCDA =
  clock?1 ->
  ( chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    DCDB [])
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
  readallowed!1 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  SKIP )
DCDB =
reset?1 ->
  ( read?0 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    SKIP [])
  read?1 -> STOP )
reset?0 ->
  ( read?0 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    DCDB [])
  read?1 -> STOP )
[]
read?0 ->
  ( reset?1 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    SKIP [])
  reset?0 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.4 ->
    readallowed!0 ->
    chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
    DCDB )
[]
read?1 -> STOP
within DCDA
DCE =
let DCEA(x) =
  length(x) == 0 & STOP []
  length(x) == 1 &
    chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC.( length(x) - length(data) - length(x) )?z ->
    DCEB(head(x), z) []
  length(x) > 1 &
    ( chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC.( length(data) - length(x) )?z ->
      DCEB(head(x), z) )
read?1 -> STOP
within clock?1 -> DCEA(data)
within ( readallowed?0 ->
  ( ( ||| x::set(data) @ x!0 -> SKIP )
  , ( chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC -> DCA )
  )
)

within DA
-- No Reads or Stores
E =
  ( reset?0 -> SKIP
[]
  reset?1 -> SKIP
  )
( clock?1 -> E )
F =
let
  PA =
  reset?1 ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  FA
[]
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  FA
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 -> FB
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 -> FC
FB =
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  FA
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 -> FD
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 -> FD
FC =
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.3 ->
  chan_link_PROC_STORAGECOMPONENT_GENERIC_SPEC ->
  FA
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.0 -> FD
[]
  chan_PROC_STORAGECOMPONENT_GENERIC_SPEC.1 -> FC
FD =
  reset?1 ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT GENERIC_SPEC.3 ->
  chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC ->
  FA
[]
  clock?1 ->
  chan_PROC_STORAGECOMPONENT GENERIC_SPEC.4 ->
  chan_link_PROC_STORAGECOMPONENT GENERIC_SPEC ->
  FD
[]
  chan_PROC_STORAGECOMPONENT GENERIC_SPEC.0 -> FD
[]
  chan_PROC_STORAGECOMPONENT GENERIC_SPEC.1 -> FD
within PA
Appendix F: Multi-Type Component Generic Specification Model Example

\[ G = \]
\[
\text{let}
\]
\[
\text{GA} =
\]
\[
\text{length(stores)} = 0 \& \text{GB(head(stores))}
\]
\[
\text{length(reads)} = 0 \& \text{GB(head(reads))}
\]
\[
\text{GB}((\_, \_, \_, \text{data})) =
\]
\[
\text{if } \text{reset}1 \rightarrow \text{GC(x, 0)}
\]
\[
\text{chan_readbits_PROC_STORAGECOMPONENT_GENERIC_SPEC.x!y} \rightarrow \text{GC(x, y)}
\]
\[
\text{chan_storebits_PROC_STORAGECOMPONENT_GENERIC_SPEC.x?z} \rightarrow \text{GC(x, z)}
\]
\[
\text{within GA}
\]
\[
\text{H} =
\]
\[
\text{F}
\]
\[
\text{G}
\]
\[
\text{within chase(A)}
\]

F.1.3 GenSpec 3: Correct Component Driving

This CSP model is used to limit a process so that it can only accept possible valid input signals, this is to enable implemented sub-type components to have the outer layer of their logic correctly driven when performing the checks and proofs. The aim for this is to check an implemented component holds true to the assumption that so long as it is driven correctly, it will correctly drive any internal components.

Figure 92: Generic Control Data Storage Specification - Correct Driving Limiter

```
channel chan_PROC_STORAGECOMPONENT_CONTROLL: \{0, 1, 2, 3\}
channel chan_link_PROC_STORAGECOMPONENT_CONTROLL

PROC_STORAGECOMPONENT_CONTROLL(clock, reset, stores, reads) =

let
A =
( C(stores)
 [\{ clock, reset \},
   \{ chan_link_PROC_STORAGECOMPONENT_CONTROLL \}
  ]
 )
B =
\{ chan_link_PROC_STORAGECOMPONENT_CONTROLL \}
-- if there are no stores or reads
E(reads)
-- Stores
C(x) =
length(x) == 0 \& B
[\]
length(x) > 0 \&
( D(head(x))
 [\{ clock, reset \},
   \{ chan_link_PROC_STORAGECOMPONENT_CONTROLL \}
  ]
 )
```

169
Appendix F: Multi-Type Component Generic Specification Model Example

C(tail(x))

-- Controll CSP taken from DataStore object
-- with an extra event to ensure all outputs happen before new inputs
D( (store, stored, databits) ) =
let
    DA =
    ( chan_PROC_STORAGECOMPONENT_CONTROL.1 ->
        (  ||| x:set(databits) @
            ( x?0 -> SKIP
            []
            x?1 -> SKIP
        )
    )
    []
    chan_PROC_STORAGECOMPONENT_CONTROL.2 ->
        (  ||| x:set(databits) @ x?0 -> SKIP )
    )
;
    ( chan_PROC_STORAGECOMPONENT_CONTROL.0 -> DA )
DB =
chan_link_PROC_STORAGECOMPONENT_CONTROL ->
    ( store?0 -> chan_PROC_STORAGECOMPONENT_CONTROL.2 ->
        ( ( reset?1 -> SKIP
            []
            reset?0 -> SKIP
        ); DC
    )
    []
    store?1 -> chan_PROC_STORAGECOMPONENT_CONTROL.1 ->
        ( reset?1 -> DC
        []
        reset?0 -> DD
    )
    []
    reset?0 ->
        ( store?1 -> chan_PROC_STORAGECOMPONENT_CONTROL.1 -> DD
        []
        store?0 -> chan_PROC_STORAGECOMPONENT_CONTROL.2 -> DC
    )
    []
    reset?1 ->
        ( ( store?1 -> chan_PROC_STORAGECOMPONENT_CONTROL.1 -> SKIP
            []
            store?0 -> chan_PROC_STORAGECOMPONENT_CONTROL.2 -> SKIP
        ); DC
    )
)

DC =
chan_PROC_STORAGECOMPONENT_CONTROL.0 -> clock?1 ->
    ( stored?0 -> DB
    []
    stored?1 -> STOP
)

DD =
chan_PROC_STORAGECOMPONENT_CONTROL.0 -> clock?1 ->
    ( stored?1 -> DB
    []
    stored?0 -> DE
)

DE =
chan_link_PROC_STORAGECOMPONENT_CONTROL ->
    ( store?0 -> chan_PROC_STORAGECOMPONENT_CONTROL.2 ->
        ( reset?1 -> DC
            []
        reset?0 -> DD
    )
    []
)
Appendix F: Multi-Type Component Generic Specification Model Example

reset?0 -> store?0 -> chan_PROC_STORAGECOMPONENT_CONTROLL.2 -> DD
reset?1 -> store?0 -> chan_PROC_STORAGECOMPONENT_CONTROLL.2 -> DC

DF =
( ( stored?0 -> DB
    [ ]
    stored?1 -> STOP
)
[ ]
{ chan_PROC_STORAGECOMPONENT_CONTROLL | }

DA
) \ { chan_PROC_STORAGECOMPONENT_CONTROLL | }
within DF

-- Reads
B(x) =
length(x) == 0 & B
[ ]
length(x) > 0 &
( F(head(x))
  [ ]
  union{{ clock, reset |},
         { chan_link_PROC_STORAGECOMPONENT_CONTROLL}
  }
  [ ]
  B(tail(x))
)

-- Controll CSP taken from DataRead object
-- with an extra event to ensure all outputs happen before new inputs
F( (read, readallowed, databits) ) =
let
FA =
  chan_link_PROC_STORAGECOMPONENT_CONTROLL ->
  ( read?x -> reset? y -> FD(x, y)
    [ ]
    reset? y -> read?x -> FD(x, y)
  )

FB =
  chan_link_PROC_STORAGECOMPONENT_CONTROLL ->
  ( read?0 -> reset? y -> FB(y)
    [ ]
    reset? y -> read?0 -> FB(y)
  )

FC =
readallowed!1 ->
( ( [ ] [ ] x: set(databits) @
      x!0 -> SKIP
      [ ]
      x!1 -> SKIP
    ); FA
  )
[ ]
readallowed!0 ->
( ( [ ] [ ] x: set(databits) @
      x!0 -> SKIP
      [ ]
      x!1 -> STOP
    ); FB
  )

FD(x, y) =
clock?1 ->
( y == 1 & FF
  [ ]
  y == 0 &
    ( x == 0 & FF
      [ ]
      x == 1 & FC
    )
  )

FE(y) =
F.1.4 GenSpec 4: Annotated Low Level Behaviour with Explicit Deadlocking

This CSP model is the one covered in section F.1.2, but with extra events added to describe conceptually what is occurring. The aim of this is to enable a link between a low level hardware model and a higher level conceptual meaning of the function the hardware is performing. The added 'id' parameter added to the process is to provide a method to distinguish between different instances of this process. The annotation events depicting the states that are entered into from how this component is driven can only be specified after the event has occurred, where as the output signals are controlled by this component and so the corresponding annotation events can be performed before outputting the signals. The reason why renaming can not be used to obtain a higher level conceptual model of what is occurring, thus the required use of extra events depicting the annotations, is because the same signal states can mean different things depending on the state of the system.
let
BA =
let
BAA =
length(stores) != length(sid) & STOP

(length(stores) == length(sid) &
(BAB(stores, sid)

[]

length(stores) = length(sid) &

(BAB(stores, sid)

[]

+ reaet. 1,
clock,
chan_link_PROCSTORACiECOMPONENTANNOTATED SPEC
chan_PROCSTORACOMPONENTANNOTATED_SPEC
[]

( chan_link_PROCSTORAGECOMPONENTANNOTATED_SPEC -> P )

\[
(BAB(x, y) =

length(x) == 1 & BB(head(x), head(y))

[]

length(x) > 1 &

(BB(head(x), head(y))

[]

union({

reset,
clock,
chan_link_PROCSTORAGECOMPONENTANNOTATED_SPEC
[]

chan_PROCSTORAGECOMPONENTANNOTATED_SPEC.y

y<\{3,4\}

})

})

[]

BAB(tail(x), tail(y))

}

within BAA
BB( (store, stored, data), id ) =

let
BBA =

(BBB

[]

clock,
chan_link_PROCSTORAGECOMPONENTANNOTATED_SPEC,
chan_PROCSTORAGECOMPONENTANNOTATED_SPEC
[]

[]

BBC

\[
chan_mid_PROCSTORAGECOMPONENTANNOTATED_SPEC[]

BBA =
let
BBBA =

(chan_mid_PROCSTORAGECOMPONENTANNOTATED_SPEC.0 ->

([], x: set(data) @

(x?0 -> SKIP

[]

x?1 -> annotation.ERROR.id -> STOP

)

[]

chan_mid_PROCSTORAGECOMPONENTANNOTATED_SPEC.1 ->

BBBB(data, 0)

)

; (clock?1 ->

chan_link_PROCSTORAGECOMPONENTANNOTATED_SPEC ->

BBBA

)
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
BBBB(x, y) =
    length(x) <= 0 & annotation.ERROR.id -> STOP

length(x) == 1 & BBBC(head(x), y)
    length(x) > 1 &
        BBBC(head(x), y)

chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.
    z<-{1, 2}

BBBBB(tail(x), y+1)

BBBC(x, y) =
    x?0 -> BBBD(y, 0)

BBBD(x, y) =
    chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 -> SKIP

chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
    annotation.DATABIT.id.x.y -> SKIP

within BBBA

BBC =
    reset?1 ->
        {
            store?0 ->
                chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
                    annotation.RESET.id -> SKIP

            store?1 ->
                chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
                    annotation.RESET.id ->
                    chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
                        SKIP

            chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
        }

    reset?0 ->
        {
            store?0 ->
                chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
                    annotation.IDLE.id ->
                    BBBD

            store?1 ->
                chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
                    annotation.STORE.id ->
                    chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
                        BBBD

            chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
        }

    store?0 -> chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
        {
            reset?0 -> annotation.IDLE.id -> SKIP

            reset?1 -> annotation.RESET.id -> SKIP
        }

    store?1 -> chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
        {
            reset?0 ->
                annotation.STORE.id ->
                chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
        }
```

174
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 -> BBE
[]
reset?1 ->
  chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
  annotation.RESET.id -> BBD
)
BBD =
clock?1 ->
  ( ( chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 -> SKIP
    [ ]
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 -> SKIP
  )
  ;
  ( stored!0 ->
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> BBC
  )
)
BBE =
clock?1 ->
  ( chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
    annotation.STORED.id ->
    stored!1 ->
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> BBC
  []
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
    annotation.NOTSTORED.id ->
    stored!0 ->
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> BBF
  )
BBF =
reset?1 ->
  ( store?0 ->
    chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
    annotation.RESET.id -> BBD
  []
  store?1 -> annotation_ERROR.id -> STOP
  )
[]
reset?0 ->
  ( store?0 ->
    chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 -> BBG
  []
  store?1 -> annotation_ERROR.id -> STOP
  )
[]
store?0 ->
  chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
  ( reset?1 -> annotation.RESET.id -> BBD
  []
  reset?0 -> BBG
  )
[]
store?1 -> annotation_ERROR.id -> STOP
BBG =
clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.NOTSTORED.id ->
  stored!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> BBF
within stored!0 -> chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> BBA
```
within BA
C =
let
CA =
let
CAA =
  length(reads) != length(rid) & STOP
[]
length(reads) == length(rid) &
  CAB(reads, rid)
[]
  [[
    reset.1,
    clock,
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC

  ]]
[]
  ( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> F )
) \[]
  [[ chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC

  ]]
CAB(x, y) =
  length(x) == 1 & CB(head(x), head(y))
[]
length(x) > 1 &
  ( CB(head(x), head(y))
  []
  union{
    reset,
    clock,
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC

  }[

  ]
  ( chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.y
    | y<=\{3,4\}
  ]

  ]
  ]
  CAB(tail(x), tail(y))
}]
within CAA
CB( (read, readallowed, data), id ) =
let
CBA =
  reset?1 ->
    ( read?0 -> SKIP
      []
      read?1 -> SKIP
    )
  ;
  ( annotation.RESET.id -> CBB )
[]
  reset?0 ->
    ( read?0 -> annotation.IDLE.id -> CBB
      []
      read?1 -> annotation.READ.id -> CBC
    )
  ]
  read?0 ->
    ( ( reset?0 -> annotation.IDLE.id -> SKIP
      []
      reset?1 -> annotation.RESET.id -> SKIP
    )
      CBB
    ]
  ]
  read?1 ->
    ( reset?0 -> annotation.READ.id -> CBC

}
Appendix F: Multi-Type Component Generic Specification Model Example

```
[ ]
reset?1 -> annotation.RESET.id -> CBB
)
CBB =
clock?1 ->
( [ chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 -> SKIP ]
[ ]
chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 -> SKIP
)
;
( readallowed!0 ->
( [ ] x:set(data) @ x!0 -> SKIP )
)
;
( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> CBA ) )
CBC =
let
CBCA =
clock?1 ->
( [ chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
 annotation.READALLOWED.id ->
 ]
readallowed!1 ->
( CBCB(data, 0)
)
;
( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
 CBA
)
)

[ ]
chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
 annotation.DONTREAD.id ->
readallowed!0 ->
( [ ] x:set(data) @ x!0 -> SKIP )
;
( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
 CBD
)
)
)
CBCB(x, y) =
length(x) == 0 & annotation.ERROR.id -> STOP
[ ]
length(x) == 1 & CBCC(head(x), y)
[ ]
length(x) > 1 &
( CBCC(head(x), y)
  [ ]
  CBCB(tail(x), y+1)
)
CBCC(x, y) =
length(x) _ 0 & annotation.ERROR.id -> x!0 -> SKIP
within CBCA
CBD =
reset?1 ->
( read0 -> annotation.RESET.id -> CBB
[ ]
  read?1 -> annotation.ERROR.id -> STOP
)
)
reset?0 ->
( read0 -> CBE
[ ]
  read?1 -> annotation.ERROR.id -> STOP
)
)
read?0 ->
( reset?1 -> annotation.RESET.id -> CBB

177
Appendix F: Multi-Type Component Generic Specification Model Example

[
  reset?0 -> CBE
]

[
  read?1 -> annotation.ERROR.id -> STOP

CBE =

  clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.DONTREAD.id ->
  readallowed:0 ->
  ( ( ||x:set(data) @ x!0 -> SKIP )
    ;
    ( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> CBD )
  )

within ( readallowed:0 ->
  ( ( ||x:set(data) @ x!0 -> SKIP )
    ;
    ( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> CBA )
  )
)

within CA

D =
  let
    DA =
    let
      DAA =
      ( ( DAB
        [ ]
          [ ]
            reset,
            clock,
            chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC
          [ ]
        [ ]
          ( chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.y
            y<-{3,4}
          )
        [ ]
      )
      )
      DAD
    [ ]
    [ ]
    reset.1,
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
    chan_readbits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
    chan_storebits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
    clock,
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC
  )

  ( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> H )

DAB =
  length(stores) = length(sid) & STOP

  length(stores) = length(sid) & DAC(stores, sid)

DAC(x, z) =
  length(x) == 1 & DB(head(x), head(z))

  length(x) > 1 &
  ( DB(head(x), head(z))
    [ ]
      union(
        [ ]
          reset,
          clock,
          chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC
        [ ]
      )
  )

178
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
{ chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.y
 | y<-{3,4}
 }

DAC(tail(x), tail(z))

DAD =
length(reads) != length(rid) & STOP
[ ]
length(reads) == length(rid) & DAE(reads, rid)

DAE(x, z) =
length(x) == 1 & DC(head(x), head(z))
[ ]
length(x) > 1 &
( DC(head(x), head(z))
[ ]
union(
[ ]
reset,
clock,
chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC
[ ]
}

within DAA
DB((store, stored, data), id) =
let
DBA =
( [ ] { chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
clock
[ ]
[ ]
DBC
)
\ { chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC | }

DBB =
let
DBBA =
( chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
  ( [ ]
   |
   x:set(data) ®
   ( x?0 -> SKIP
   [ ]
   x?1 -> annotation.ERROR.id -> STOP
   )
   )
 )
;
( clock?1 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DBBA
 )
)

chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
( DBBB(data)
  ;
  DBBA
 )

DBBB(x) =
length(x) == 0 & annotation.ERROR.id -> STOP
[ ]
```
length(x) == 1 & DBBC(head(x), length(data) - 1)
[]
length(x) > 1 &
{ DBBC(head(x), length(data) - length(x))
 [] { chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2,
 clock.1
 }
 []
 DBBB(tail(x))
 }
DBBC(x, y) =
{ ( x?0 ->

chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
chan_storebits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.y.0 ->
annotation.DATABIT.id.y.0 ->
clock?1 ->
SKIP
[]
clock?1 -> SKIP

[]
x?1 ->
{}
chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
chan_storebits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.y.1 ->
annotation.DATABIT.id.y.1 ->
clock?1 ->
SKIP
[]
clock?1 -> SKIP

)
;
( chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> SKIP )
)
within DBBS
DBC =
reset?1 ->
{ { store?0 ->
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
 SKIP
 []
 store?1 ->
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
 SKIP
 }
 ( annotation.RESET.id -> DBD )
 }
[]
reset?0 ->
{ store?0 ->
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
 annotation.IDLE.id ->
 DBD
 []
 store?1 ->
 chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
 annotation.STORE.id ->
 chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2 ->
 DBS
 }
[]
store?0 -> chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0 ->
{ ( reset?0 -> annotation.IDLE.id -> SKIP

}
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext

[[]
  reset?1 \rightarrow \text{annotation.RESET.id} \rightarrow \text{SKIP }
}
;
DBD
)
[]

store?1 \rightarrow \text{chan_mid_PROC_STORCGEPONENT_ANNOTATED_SPEC.1} \rightarrow
( reset?0 \rightarrow
  \text{chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0} \rightarrow
  \text{annotation.STORE.id} \rightarrow
  \text{chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.2} \rightarrow
  \text{DBE }
[]

reset?1 \rightarrow \text{annotation.RESET.id} \rightarrow \text{DBD }

DBD =
clock?1 \rightarrow
( ( \text{chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3} \rightarrow \text{SKIP }
[]
  \text{chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4} \rightarrow \text{SKIP }
)
;
( \text{stored!0} \rightarrow
  \text{chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC} \rightarrow
  \text{DBC }
)
)

DBE =
clock?1 \rightarrow
( \text{chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3} \rightarrow \text{STORED.id} \rightarrow
  \text{stored!1} \rightarrow
  \text{chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC} \rightarrow
  \text{DBC }
[]
  \text{chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4} \rightarrow \text{NOTSTORED.id} \rightarrow
  \text{stored!0} \rightarrow
  \text{chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC} \rightarrow
  \text{DBF }
)

DBF =
reset?1 \rightarrow
( \text{store?0} \rightarrow
  \text{chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0} \rightarrow
  \text{annotation.RESET.id} \rightarrow
  \text{DBD }
[]

\text{store?1} \rightarrow \text{annotation.ERROR.id} \rightarrow \text{STOP }
)
]

[]

reset?0 \rightarrow
( \text{store?0} \rightarrow
  \text{chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0} \rightarrow
  \text{DBG }
[]

\text{store?1} \rightarrow \text{annotation.ERROR.id} \rightarrow \text{STOP }
)
]

\text{store?0} \rightarrow \text{chan_mid_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.0} \rightarrow
( \text{reset?1} \rightarrow \text{annotation.RESET.id} \rightarrow \text{DBD }
[]

\text{reset?0} \rightarrow \text{DBG }
)
]

\text{store?1} \rightarrow \text{annotation.ERROR.id} \rightarrow \text{STOP }

DBG =
```

181
Appendix F: Multi-Type Component Generic Specification Model Example

clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.NOTSTORED.id ->
stored?0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
DBF
within ( stored?0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DBA)
)
DC( (read, readallowed, data), id ) =
let
DCA =
  reset?1 ->
  ( (read?0 -> SKIP
  ([]
    read?1 -> SKIP
  )
  ;
  (annotation.RESET.id -> DCB)
  )
  []
reset?0 ->
  ( read?0 -> annotation.IDLE.id -> DCB
  []
  read?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
  annotation.READ.id ->
DCC
  )
  []
read?0 ->
  ( ( read?0 -> annotation.IDLE.id -> SKIP
  []
  reset?1 -> annotation.READ.id -> SKIP
  )
  ;
  DCB
  )
  []
read?1 ->
  ( reset?0 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 ->
  annotation.READ.id ->
DCC
  []
  reset?1 -> annotation.RESET.id -> DCB
  )
DCB =
clock?1 ->
  ( (chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 -> SKIP
  []
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 -> SKIP
  )
  ;
  (readallowed?0 ->
  ( || | x:set(data) @ x!0 -> SKIP)
  )
  ;
  (chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC -> DCA)
  )
DCC =
  ( ( DCD
      []
      [ ] chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
      readallowed,
      chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC,
      reset,
      read,
  )
  )
Appendix F: Multi-Type Component Generic Specification Model Example

clock
{]
    DCE


; DCA

let

DCD =

DCDA =
clock?1 ->
{ chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.DONTREAD.id ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DCDB

[]
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
  annotation.READALLOWED.id ->
  readallowed!1 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  SKIP
}

DCDB =

reset?1 ->
{ read?0 ->
  annotation.RESET.id ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  SKIP

[]
  read?1 -> annotation.ERROR.id -> STOP

}

reset?0 ->
{ read?0 ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.DONTREAD.id ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DCDB

[]
  read?1 -> annotation.ERROR.id -> STOP

}

read?0 ->
{ reset?1 ->
  annotation.RESET.id ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  SKIP

[]
  reset?0 ->
  clock?1 ->
  chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  annotation.DONTREAD.id ->
  readallowed!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DCDB

[]
  read?1 -> annotation.ERROR.id -> STOP
within DCDA
DCE =
let
  DCEA(x) =
    length(x) == 0 & annotation.ERROR.id -> STOP
    []
    length(x) == 1 &
    {chan_readbits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.(
      length(data) - length(x)
    )?z ->
      DCEB(head(x), z, (length(data) - length(x)))
    )
    []
    length(x) > 1 &
    {chan_readbits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.(
      length(data) - length(x)
    )?z ->
      DCEB(head(x), z, (length(data) - length(x)))
    )
    []
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC, readallowed, chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC, reset, read, clock
    }
    ]
    DCEA(tail(x))
)

DCEB(x, y, z) =
chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
  readallowed!0 ->
  x!0 ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  DCEC(x)
[]
chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
  readallowed!1 ->
  annotation.DATABIT.id.z.y ->
  x!y ->
  chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
  SKIP
DCEC(x) =
  reset?1 ->
  (read?0 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.3 ->
    readallowed!0 ->
    x!0 ->
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
    SKIP
    []
    read?1 -> STOP
  )
  []
reset?0 ->
  (read?0 ->
    clock?1 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.4 ->
    readallowed!0 ->
    x!0 ->
    chan_link_PROC_STORAGECOMPONENT_ANNOTATED_SPEC ->
    DCEC(x)
    []
    read?1 -> STOP
  )
Appendix F: Multi-Type Component Generic Specification Model Example

\[
\begin{align*}
&\text{read?0} \rightarrow \\
&\quad (\text{reset?1} \rightarrow \\
&\quad \quad \text{clock?1} \rightarrow \\
&\quad \quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{readallowed!0} \rightarrow \\
&\quad \quad \quad \text{x:10} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \\
&\quad \quad \quad \text{SKIP} \\
&\quad ) \\
&\text{reset?0} \rightarrow \\
&\quad \text{clock?1} \rightarrow \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.4} \rightarrow \\
&\quad \quad \text{readallowed!0} \rightarrow \\
&\quad \quad \text{x:10} \rightarrow \\
&\quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \\
&\quad \quad \text{DCEC(x)}
\end{align*}
\]

\[
\begin{align*}
&\text{read?1} \rightarrow \text{STOP} \\
&\quad \text{within} \quad \text{clock?1} \rightarrow \text{DCEA(data)} \\
&\quad \text{within} \quad \text{( readallowed!0 } \rightarrow \\
&\quad \quad \text{let } \\
&\quad \quad \quad \text{FA} \rightarrow \text{reset?1} \rightarrow \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FA} \\
&\quad \quad \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FA} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.0} \rightarrow \text{FB} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.1} \rightarrow \text{PC} \\
&\quad \text{FB} = \\
&\quad \quad \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FA} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.0} \rightarrow \text{FD} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.1} \rightarrow \text{FD} \\
&\quad \text{PC} = \\
&\quad \quad \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FA} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.0} \rightarrow \text{FD} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.1} \rightarrow \text{FD} \\
&\quad \text{FD} = \\
&\quad \quad \text{reset?1} \rightarrow \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.3} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FA} \\
&\quad \quad \text{clock?1} \rightarrow \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.4} \rightarrow \\
&\quad \quad \quad \text{chan\_link}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC} \rightarrow \text{FD} \\
&\quad \quad \text{chan}_\text{PROC\_STORAGECOMPONENT\_ANNOTATED\_SPEC.0} \rightarrow \text{FD}
\end{align*}
\]
Appendix F: Multi-Type Component Generic Specification Model Example

F.1.5 GenSpec 5: Annotating the Outer Layer

This CSP model is used to annotate the outer layer of an implemented sub-type of this component. The process allows correct input and output signals to be able to annotate and describe that an error has occurred. No internal choice is used, as this should not restrict or control a process, but only annotate what is occurring.

Figure 94: Generic Data Storage Annotate Outer Layer

channel chan_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.1 -> FD
within FA
G =
let
GA =
   length(stores) != 0 & GB(head(stores))
[0]
length(reads) != 0 & GB(head(reads))
GB (__, __, data) =
   [[ { reset, 1 } ] x: [0 .. (length(data) - 1)] ® GC(x, 0)
   GC(x, y) =
   reset? 1 -> GC(x, 0)
[1]
chan_readbits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.x! y -> GC(x, y)
[1]
chan_storebits_PROC_STORAGECOMPONENT_ANNOTATED_SPEC.x? z -> GC(x, z)
within GA
H =
F
{ [ { reset, 1 } ]
G
within chase(A)
C(tail(x), tail(y))

-- Controll CSP taken from DataStore object
-- with an extra event to ensure all outputs happen before new inputs
D((store, stored, databits), id) =
let
DA =
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.1 -> DH(databits)
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
(x: set(databits) ♩
x?0 -> SKIP
x?1 -> annotation.ERROR.id -> STOP
)
)
;
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.0 -> DA)
DH(x) =
length(x) == 0 & annotation.ERROR.id -> STOP
length(x) == 1 & DI(head(x), length(databits)-1)
length(x) > 1 &
(DI(head(x), length(databits)-length(x))
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.y | y∈{4, 5})]
DH(tail(x))
)

DI(x, y) =
x?1 ->
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.4 ->
annotation.DATABIT.id.y.1 ->
SKIP
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.5 -> SKIP
)
]
]
x?0 ->
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.4 ->
annotation.DATABIT.id.y.0 ->
SKIP
]
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.5 -> SKIP
)

DB =
chan_link_PROC_STORAGECOMPONENT_ANNOTATE_OUTER ->
(store?0 -> chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
(reset?1 -> annotation.RESET.id -> SKIP
(reset?0 -> annotation.IDLE.id -> SKIP
;)
);
DC
)[
]
]
store?1 -> chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.1 ->
(reset?1 ->
annotation.RESET.id ->
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.5 ->
DC
[]
(reset?0 ->
annotation.STORE.id ->
chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.4 ->
DD
)
Appendix F: Multi-Type Component Generic Specification Model Example

```
reset?0 ->
  ( store?1 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.1 ->
    annotation.STORE.id ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.4 ->
    DD
  )
reset?0 ->
  ( store?1 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
    annotation.IDLE.id ->
    DC
  )
reset?1 ->
  ( store?1 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.1 ->
    annotation.RESET.id ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.5 ->
    SKIP
  )
reset?0 ->
  ( store?0 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
    annotation.RESET.id ->
    SKIP
  )
  DC
)

DC =
  chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.0 -> clock?1 ->
  ( stored?0 -> DB
    []
    stored?1 -> annotation.ERROR.id -> STOP
  )

DD =
  chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.0 -> clock?1 ->
  ( stored?1 -> annotation.STORED.id -> DB
    []
    stored?0 -> annotation.NOTSTORED.id -> DE
  )

DE =
  chan_LINK_PROC_STORAGECOMPONENT_ANNOTATE_OUTER ->
  ( store?1 -> annotation.ERROR.id -> STOP
    []
    store?0 ->
    chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
    reset?y ->
    DG(y)
    []
    reset?y ->
    ( store?0 ->
      chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER.2 ->
      DG(y)
      []
      store?1 -> annotation.ERROR.id -> STOP
    )
  )

DF =
  ( ( stored?0 -> DB
    []
    stored?1 -> annotation.ERROR.id -> STOP
  )
   | | chan_PROC_STORAGECOMPONENT_ANNOTATE_OUTER |} |]
DA
DG(y) =
```

188
Appendix F: Multi-Type Component Generic Specification Model Example

```

y == 1 & annotation.RESET.id -> DC
[]

y == 0 & DD

within DF

-- Reads

E(x, y) =
  length(x) == 0 & B
[]

length(x) > 0 &
  ( F(head(x), head(y))
    [ | union( { [ | clock, reset |],
        { chan_link_PROStORAGECOMPONENT_ANNOTATE_OUTER } ]
    )
    [ ]
    E(tail(x), tail(y))
  )

-- Control CSP taken from DataRead object
-- with an extra event to ensure all outputs happen before new inputs

F( (read, readallowed, databits), id ) =

let

  FA =
    chan_link_PROStORAGECOMPONENT_ANNOTATE_OUTER ->
    ( read?x -> reset? y -> FD(x, y)
      []
      reset? y -> read?x -> FD(x, y)
    )

  FB =
    chan_link_PROStORAGECOMPONENT_ANNOTATE_OUTER ->
    ( read?0 ->
      ( reset?1 -> annotation.RESET.id -> clock?1 -> FF
        []
        reset?0 -> clock?1 -> FC )
      [ ]
      reset?1 -> read?0 -> annotation.RESET.id -> clock?1 -> FF
      [ ]
      reset?0 -> read?0 -> clock?1 -> FC )

  FC =
    readallowed!1 -> annotation.READALLOWED.id ->
    ( FG(databits)
      ;
    )
  FB

  readallowed!0 -> annotation.DONTREAD.id ->
  ( ( | | x: set(databits) @
    x!0 -> SKIP
    [ ]
    x!1 -> annotation.ERROR.id -> STOP
  )
  ;
  FB

  FD(x, y) =
    y == 1 & annotation.RESET.id -> clock?1 -> FF
    []

    y == 0 &
    ( x == 0 & annotation.IDLE.id -> clock?1 -> FF
      []
      x == 1 & annotation.READ.id -> clock?1 -> FC
    )

  FE(y) =
    y == 1 & annotation.RESET.id -> clock?1 -> FF
    []

    y == 0 & clock?1 -> FC

  FF =
    readallowed!1 -> annotation.ERROR.id -> STOP
```

189
Appendix F: Multi-Type Component Generic Specification Model Example

F.1.6 GenSpec 6: Clock Cycle Higher Generic Specification

This CSP model is an annotation only clock cycle based higher conceptual specification. It is used as a comparison for the extracted annotations from the annotated low level hardware models.

Figure 95: Generic Data Storage Annotation Specification

let
A = length(sid) != 0 and length(rid) != 0 & SKIP
B = length(sid) != 0 and length(rid) != 0 & B
C = length(sid) != 0 and length(rid) != 0 & C
D = length(sid) != 0 and length(rid) != 0 & D

let
BA = annotation.RESET, chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2, chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3, chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC

x: set(sid) @ BB(x)

channel chan_mid_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC: {0..2}
channel chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC
channel chan_readbits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC: {0..4}
channel chan_storebits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC: {0..2}, {0, 1}
PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC(bitLength, sid, rid) = let

within FP
within chase(A)
Appendix F: Multi-Type Component Generic Specification Model Example

```
annotation.RESET
|
[]
BD
BB(x) =
( ( | y:set(sid) \* annotation.RESET.y \rightarrow SKIP ) ; BB(x) )
[]
anotation.IDLE.x ->
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  ( ( chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 \rightarrow SKIP
    []
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 \rightarrow SKIP
  )
   ;
  ( chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC \rightarrow BB(x) )
)
[]
anotation.STORE.x ->
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.0 ->
  ( ( | y:0..(bitLength-1)} \* annotation.DATABIT.x.y.0 \rightarrow SKIP
    []
    annotation.DATABIT.x.y.1 \rightarrow SKIP
  )
   ;
  ( chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC \rightarrow
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 \rightarrow
    annotation.STORED.x ->
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC \rightarrow
    BB(x)
    []
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 \rightarrow
    annotation.NOTSTORED.x ->
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC \rightarrow
    BC(x)
  )
)
BC(x) =
( ( | y:set(sid) \* annotation.RESET.y \rightarrow SKIP ) ; BB(x) )
[]
chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 ->
  annotation.NOTSTORED.x ->
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  BC(x)
-- Storage
BD =
chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  BD
[]
chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.0 -> BE
[]
( ( | y:set(sid) \* annotation.RESET.y \rightarrow SKIP ) ; BD )
BE =
chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  BD
[]
chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.0 -> BF
BF =
( ( | y:set(sid) \* annotation.RESET.y \rightarrow SKIP ) ; BD )
[]
chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
```
Appendix F: Multi-Type Component Generic Specification Model Example

```
chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC.3 ->
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC ->
BF

[]

chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC.0 -> BF
within ( BA \ { chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC, 
chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC }
)

C =
let
CA =
[]
{ annotation.RESET, 
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC
}
[] x:set(rid) @ CB(x)
CB(x) =
{ ( { y:set(rid) @ annotation.RESET.y -> SKIP ); CB(x) )
]
annotation.IDLE.x ->
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC ->
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC ->
CB(x)

[]
annotation.READ.x ->
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC ->
CC(x)
CC(x) =
annotation.READALLOWED.x ->
{ ( { y:0...(bitLength-1)) @ annotation.DATABIT.x.y.0 -> SKIP )
; 
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC -> CB(x) )
}
within CA \ { chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC }

D =
let
DA =
{ ( { union({
{ annotation.RESET, 
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC
}
{ chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC.y 
| y<-{2, 3}
}
}
|] x:set(sid) @ DB(x)

[ union({
{ annotation.RESET, 
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC
}
{ chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC.y 
| y<-{2, 3}
}
}

)}
{ union({
{ annotation.RESET, 
chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC
}
{ chan_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC.y 
| y<-{2, 3}
}
}
]| x:set(rid) @ DE(x)

})

] chan_link_PROC_STORAGECOMPONENT_HIGHER_outer_SPEC,
```

192
Appendix F: Multi-Type Component Generic Specification Model Example

\[
\begin{align*}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{chan\_storebits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{chan\_readbits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{annotation.\_RESET} \\
\{ \\
\} \\
\text{DJ} \\
) \{ \\
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{chan\_storebits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC,} \\
\text{chan\_readbits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC} \\
\} \\
\text{-- Store} \\
\text{DB(x) =} \\
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC} \rightarrow \\
\{ \text{ ( ( } \text{ y:union(set(sid), set(rid)) } \text{ ) } \text{ ) } \text{ annotation.\_RESET. y } \rightarrow \text{ SKIP} \}
\}
\text{annotation.\_IDLE. x } \rightarrow \\
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC} \rightarrow \\
\{ \text{ ( chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.2 } \rightarrow \text{ SKIP} \}
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.3 } \rightarrow \text{ SKIP} \\
; \text{ DB(x) } \\
\}
\} \\
\text{annotation.\_STORE. x } \rightarrow \\
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.0 } \rightarrow \\
\{ \text{ ( ( } \text{ y: set(0..(bitLength-1)) } \text{ ) } \text{ ) } \text{ annotation.DATABIT. x. y. 0 } \rightarrow \text{ SKIP} \}
\}
\text{chan\_storebits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC. y. 0 } \rightarrow \text{ SKIP} \\
\}
\text{annotation.DATABIT. x. y. 1 } \rightarrow \\
\text{chan\_storebits\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC. y. 1 } \rightarrow \text{ SKIP} \\
\}
\}
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC } \rightarrow \\
\{ \text{ ( ( } \text{ y:union(set(sid), set(rid)) } \text{ ) } \text{ annotation.\_RESET. y } \rightarrow \text{ SKIP} \}
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.2 } \rightarrow \text{ SKIP} \\
; \text{ DB(x) } \\
\}
\} \\
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC } \rightarrow \\
\{ \text{ ( ( } \text{ y:union(set(sid), set(rid)) } \text{ ) } \text{ annotation.\_RESET. y } \rightarrow \text{ SKIP} \}
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.3 } \rightarrow \text{ annotation.\_NOTSTORED. x } \rightarrow \text{ SKIP} \\
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC } \rightarrow \\
\{ \text{ ( ( } \text{ y:union(set(sid), set(rid)) } \text{ ) } \text{ annotation.\_RESET. y } \rightarrow \text{ SKIP} \}
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.3 } \rightarrow \text{ annotation.\_NOTSTORED. x } \rightarrow \text{ SKIP} \\
\}
\text{DB(x) =} \\
\text{chan\_link\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC } \rightarrow \\
\{ \text{ ( ( } \text{ y:union(set(sid), set(rid)) } \text{ ) } \text{ annotation.\_RESET. y } \rightarrow \text{ SKIP} \}
\}
\text{chan\_PROC\_STORAGECOMPONENT\_HIGHER\_OUTER\_SPEC.2 } \rightarrow \text{ SKIP} \\
; \text{ DB(x) } \\
\}
\}
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
-- Reads
DE(x) =
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    ( ( ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP
        )
        ;
        DE(x)
    )
)

annotation.IDLE.x ->
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    ( ( chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 -> SKIP
        [ ]
        chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 -> SKIP
    )
    ;
    DE(x)
)

[]

annotation.READ.x ->
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.1 ->
    ( ( [] union
        ( chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.y,
            annotation.READALLOWED.x,
            annotation.DONTREAD.x,
            chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC
            | y<-{2, 3}
        )
        )
        []
        y:{0..(bitLength-1)} @
        ( DG(x, y, 0)
    chan_readbits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.y.0 ->
        DG(x, y, 0)
    chan_readbits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.y.1 ->
        DG(x, y, 1)
    )
)

[]

union({ chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.y,
            annotation.READALLOWED.x,
            annotation.DONTREAD.x,
            chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC
            | y<-{2, 3}
        )
        []
        [ ]
        DF(x)
)

; DE(x)
)

DF(x) =
    ( ( ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
    [ ]
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    ( chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
        annotation.READALLOWED.x ->
        SKIP
    [ ]
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 ->
        annotation.DONTREAD.x ->
        chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
        DF(x)
```

194
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
DG(x, y, z) =
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    ( chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
      annotation.READALLOWED.x ->
      annotation.DATABIT.x.y.z ->
      SKIP []
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 ->
      annotation.DONTREAD.x ->
      chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    DI(x)
  )
DI(x) =
  ( ( ||| y:union(set(sid), set(rid)) * annotation.RESET.y -> SKIP ) []
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.3 ->
    annotation.DONTREAD.x ->
    chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
  DI(x)
  )
-- Storage
DJ =
  ( [ [] annotation.RESET ] ] ) x:{0..(bitLength-1)} * DK(x, 0) ]
  [ [] annotation.RESET [] ] ]
DL
DK(x, y) =
  ( ( ||| y:union(set(sid), set(rid)) * annotation.RESET.y -> SKIP )
    DK(x, 0) ]
  [ ]
    chan_readbits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.x.y -> DK(x, y)
  [ ]
    chan_storebits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.x.0 -> DK(x, 0)
  [ ]
    chan_storebits_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.x.1 -> DK(x, 1)
DL =
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    ( chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
      chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
    DL []
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.0 -> DM []
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.1 -> DN []
    ( ( ||| y:union(set(sid), set(rid)) * annotation.RESET.y -> SKIP )
      DL []
    )
  )
DM =
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
  DL []
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.0 -> DO []
  chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.1 -> DO
DN =
  chan_link_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC ->
    chan_PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC.2 ->
  DL []
```

195
F.2 Assertions: Linking the Models Together

To ensure consistency between all the models for a generic component, several assertions have to be proven. The consistency between the models is required because the proof of an implemented component can utilise several of these specifications.

F.2.1 GenSpec Assertion 1: Initial Deadlock-Free Check

This initial deadlock-free check (see Figure 96) of GenSpec 1 (see section F.1.1) is to provide a base comparison for future deadlock-free checks and trace refinements.
Figure 96: Example of GenSpec Assertion 1 for Data Storage Specification

F.2.2 GenSpec Assertion 2: GenSpec 2 Contains GenSpec 1 Behaviour

This assertion (see Figure 97) demonstrates that GenSpec 2 (see section F.1.2) contains all the behaviour dictated by GenSpec 1 (see section F.1.1), although this assertion allows GenSpec 2 to provide extra behaviours.
Appendix F: Multi-Type Component Generic Specification Model Example

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}
channel chan4 : {0, 1}
channel chan5 : {0, 1}
channel chan6 : {0, 1}
channel chan7 : {0, 1}
channel chan8 : {0, 1}
channel chan9 : {0, 1}
channel chan10 : {0, 1}
channel chan11 : {0, 1}
channel chan12 : {0, 1}
channel chan13 : {0, 1}
channel chan14 : {0, 1}
channel chan15 : {0, 1}
channel chan16 : {0, 1}
channel chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC5 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 ]
PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  < (chan6, chan7, <chan8,chan9> )
>,
< (chan10, chan11, <chan12,chan13> ),
< (chan14, chan15, <chan16,chan17> )
>
)
alpha_PROC6 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 ]
PROC6 = PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  < (chan6, chan7, <chan8,chan9> )
>,
< (chan10, chan11, <chan12,chan13> ),
< (chan14, chan15, <chan16,chan17> )
>
)
-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC6 \ {internalChoice} )
GEN_SPEC2 = ( PROC5 \ {internalChoice} )
-- Check GenSpec 2 contains the behaviour of GenSpec 1
assert GEN_SPEC2 [T= GEN_SPEC1

Figure 97: Example of GenSpec Assertion 2 for Data Storage Specification

F.2.3 GenSpec Assertion 3: GenSpec 3 Compatible with GenSpec 2
This assertion (see Figure 98) demonstrates that the GenSpec 3 controlling specification (see section F.1.3) does not introduce any new behaviour to the specifications it is being run in parallel with. This still leaves the possibility of it limiting the events that can occur, but does not guarantee any properties regarding this.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = { | chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
chan3, chan6 |
PROC4 = PROC_STORAGECOMPONENT_CONTROLL( chan0, chan1,
< (chan2, chan3, <chan4,chan5> ),
<chan6, chan7, <chan8,chan9> )
>,
< (chan10, chan11, <chan12,chan13> ),
< (chan14, chan15, <chan16,chan17> )
>
)
alpha_PROC5 = { | chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
chan3, chan6 |
PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC( chan0, chan1,
< (chan2, chan3, <chan4,chan5> ),
<chan6, chan7, <chan8,chan9> )
>,
< (chan10, chan11, <chan12,chan13> ),
< (chan14, chan15, <chan16,chan17> )
>
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC4
GEN_SPEC2 = ( PROC5 \ {internalChoice} )

-- GenSpec3 limited by the control specification GenSpec3
GEN_SPEC2_WITH_CONTROLL = GEN_SPEC2 [ | alpha_PROC2 | ] GEN_SPEC3

-- Check GenSpec2 contains the behaviour of GenSpec2 limited by GenSpec3
assert GEN_SPEC2 [|= GEN_SPEC2_WITH_CONTROLL

Figure 98: Example of GenSpec Assertion 3 for Data Storage Specification
F.2.4 GenSpec Assertion 4: GenSpec 3 Removes Deadlock from GenSpec 2

This assertion (see Figure 99) demonstrates that GenSpec 3 (see section F.1.3) removes the possibility of driving the component it is run in parallel with, incorrectly. This does not dictate that GenSpec 3 does not remove a correctly driving option.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
    chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
    chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
    chan3, chan6 }

PROC4 = PROC_STORAGECOMPONENT_CONTROLL(
    chan0,
    chan1,
    < (chan2, chan3, <chan4, chan5> ),
    (chan6, chan7, <chan8, chan9> )
>
, (chan10, chan11, <chan12, chan13> ),
    (chan14, chan15, <chan16, chan17> )
>
)

alpha_PROC5 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
    chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
    chan3, chan6 }

PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
    chan0,
    chan1,
    < (chan2, chan3, <chan4, chan5> ),
    (chan6, chan7, <chan8, chan9> )
>
, (chan10, chan11, <chan12, chan13> ),
    (chan14, chan15, <chan16, chan17> )
>
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC4
GEN_SPEC2 = ( PROC5 \ {internalChoice} )

-- GenSpec2 limited by the control specification GenSpec3
GEN_SPEC2_WITH_CONTROL = GEN_SPEC2 \ {alpha_PROC4 } \ GEN_SPEC3

-- Check that GenSpec3 removes the incorrect driving options from GenSpec2
assert GEN_SPEC2_WITH_CONTROL : {deadlock free [F]}

Figure 99: Example of GenSpec Assertion 4 for Data Storage Specification

F.2.5 GenSpec Assertions 5: GenSpec 3 Removes Only Incorrect Driving

These assertions (see Figure 100) demonstrate that GenSpec 3 limits the process it is run in parallel with, such that it only allows correct driving signals. These assertions also
demonstrates that GenSpec 3 does not introduce any extra behaviours and does not remove any correct driving options, it is achieved through proving that GenSpec 3 run in parallel with GenSpec 2 is indistinguishable to GenSpec 1.
Appendix F: Multi-Type Component Generic Specification Model Example

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
chan3, chan6 }

PROC4 = PROC_STORAGECOMPONENT_CONTROLL(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>,
 < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
)

alpha_PROC5 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
chan3, chan6 }

PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>,
 < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
)

alpha_PROC6 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
chan3, chan6 }

PROC6 = PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>,
 < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC6 \ {internalChoice} )
GEN_SPEC2 = ( PROC5 \ {internalChoice} )
GEN_SPEC3 = PROC4

-- GenSpec2 limited by the control specification GenSpec3
GEN_SPEC2_WITH_CONTROLL = GEN_SPEC2 \ { alpha_PROC4 } \ GEN_SPEC3

-- Check that GenSpec3 in parallel with GenSpec2 is indistinguishable to
-- GenSpec1
assert GEN_SPEC1 [FD= GEN_SPEC2_WITH_CONTROLL
assert GEN_SPEC2_WITH_CONTROLL [FD= GEN_SPEC1

Figure 100: Example of GenSpec Assertions 5 for Data Storage Specification
Appendix F: Multi-Type Component Generic Specification Model Example

F.2.6 GenSpec Assertions 6: Properties of the Annotation Events

These assertions (see Figure 101) demonstrate that the annotation event contained within GenSpec 4 do not introduce extra behaviours, but are only used to conceptually describe what is occurring. This is achieved through hiding the annotation events, and proving that the resultant process is indistinguishable to the GenSpec 2 model.

```
-- channel declarations
allowedIDS = {0 .. 4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.[0 .. 2].[0, 1]
channel annotation : STATES
channel internalChoice
channel chan0 : [1]
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
  chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC1 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 } |
PROC1 = PROC_STORAGECOMPONENT_ANNOTATED_SPEC(
  chan0,
  chan1,
  ( chan2, chan3, <chan4, chan5> ),
  (chan6, chan7, <chan8, chan9> )
)

PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
  chan0,
  chan1,
  ( chan2, chan3, <chan4, chan5> ), (chan6, chan7, <chan8, chan9> ) >,
  ( chan10, chan11, <chan12, chan13> ), (chan14, chan15, <chan16, chan17> ) >
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = ( PROC1 \ {internalChoice} )
GEN_SPEC4 = ( PROC5 \ {internalChoice} )
-- GenSpec4 with the annotations hidden
GEN_SPEC4_NO_ANNOTATIONS = ( GEN_SPEC4 \ { annotations } )

-- Check that GenSpec3 in parallel with GenSpec2 is indistinguishable to
-- GenSpec1
assert GEN_SPEC4_NO_ANNOTATIONS [FD= GEN_SPEC2
assert GEN_SPEC2 [FD= GEN_SPEC4 NO_ANNOTATIONS
```

Figure 101: Example of GenSpec Assertions 6 for Data Storage Spec
F.2.7 GenSpec Assertion 7: GenSpec 3 Compatible with GenSpec 4

This assertion (see Figure 102) proves that the control process GenSpec 3 does not add extra behaviours to the annotated generic specification GenSpec 4.

```
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES

channel internalChoice
  channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
  chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, 
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, 
  chan3, chan6 } 

PROC4 = PROC_STORAGECOMPONENT_CONTROL( 
  chan0, 
  chan1, 
  <(chan2, chan3, <chan4,chan5> ), 
  (chan6, chan7, <chan8,chan9> ) 
  >, 
  <(chan10, chan11, <chan12,chan13> ), 
  (chan14, chan15, <chan16,chan17> ) 
  > )

alpha_PROC1 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, 
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, 
  chan3, chan6 } 

PROC1 = PROC_STORAGECOMPONENT_ANNOTATED_SPEC( 
  chan0, 
  chan1, 
  <(chan2, chan3, <chan4,chan5> ), 
  (chan6, chan7, <chan8,chan9> ) 
  >, 
  <1, 2 >, 
  <(chan10, chan11, <chan12,chan13> ), (chan14, chan15, <chan16,chan17> ) >, 
  <3, 4 > )

-- Hide the internalChoice event to ensure that processes has internal choice 
-- performing correctly if needed.
GEN_SPEC3 = PROC4
GEN_SPEC4 = ( PROC1 \ {internalChoice} )

-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROL = GEN_SPEC4 \ alpha_PROC4 \ GEN_SPEC3

-- Check GenSpec4 contains the behaviour of GenSpec4 limited by GenSpec3
assert GEN_SPEC4 (?= GEN_SPEC4_WITH_CONTROL
```

Figure 102: Example of GenSpec Assertion 7 for Data Storage Specification
F.2.8 GenSpec Assertion 8: GenSpec 3 Removes Deadlock From GenSpec 4

This assertion (see Figure 103) proves that the control process GenSpec 3 removes the explicitly defined deadlocks from the annotated generic specification GenSpec 4.

```plaintext
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 }
PROC4 = PROC_STORAGECOMPONENT_CONTROLL(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>),
  < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
>
)
alpha_PROC1 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 }
PROC1 = PROC_STORAGECOMPONENT_ANNOTATED_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>),
  <1, 2 >,
  < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
>),
  <3, 4 >
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC4
GEN_SPEC4 = ( PROC1 \ {internalChoice} )
-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROL = GEN_SPEC4 \ [ alpha_PROC4 ] \ GEN_SPEC3

-- Check that GenSpec3 removes the incorrect driving options from GenSpec4
assert GEN_SPEC4_WITH_CONTROL : [deadlock free [F]]
```

Figure 103: Example of GenSpec Assertion 8 for Data Storage Specification
Appendix F: Multi-Type Component Generic Specification Model Example

F.2.9 GenSpec Assertions 9: GenSpec 4 is an Annotated GenSpec 2

These assertions (see Figure 104) demonstrate that if the annotation events contained within GenSpec 4 are hidden, then GenSpec is indistinguishable to GenSpec 2.

```plaintext
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10, chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROCS = [{ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 }
PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4, chan5> ),
  (chan6, chan7, <chan8, chan9> )
>,
  < (chan10, chan11, <chan12, chan13>,
    (chan14, chan15, <chan16, chan17> )
  >)
alpha_PROC1 = [{ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 }]
PROC1 = PROC_STORAGECOMPONENT_ANNOTATED_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4, chan5> ),
  (chan6, chan7, <chan8, chan9> )
>,$
  <1, 2 >,
  < (chan10, chan11, <chan12, chan13>,
    (chan14, chan15, <chan16, chan17> )
  >,
  <3, 4 >)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = ( PROC5 \ {internalChoice} )
GEN_SPEC4 = ( PROC1 \ {internalChoice} )
-- GenSpec with hidden annotation events
GEN_SPEC4_NO_ANNOTATIONS = ( GEN_SPEC4 \ {| annotation |} )

-- Check that GenSpec with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC4_NO_ANNOTATIONS (FD= GEN_SPEC2
assert GEN_SPEC2 (FD= GEN_SPEC4_NO_ANNOTATIONS

Figure 104: Example of GenSpec Assertions 9 for Data Storage Specification
```
F.2.10 GenSpec Assertion 10: GenSpec 6 Deadlock-Free

This assertion (see Figure 105) deadlock-free checks GenSpec 6 which helps to provide a base comparison for future deadlock-free checks and trace refinements for the annotations.

```
-- Channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES

-- Higher Outer Spec
GEN_SPEC6 = PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC(2, <1, 2>, <3, 4>)

-- Check that GenSpec6 is deadlock-free
assert GEN_SPEC6 : [deadlock free [F]]
```

Figure 105: Example of GenSpec Assertion 10 for Data Storage Specification

F.2.11 GenSpec Assertions 11: GenSpec 5 Annotates Outer Level Correctly

These assertions (see Figure 106) demonstrate that GenSpec 5 will annotate the outer level of a correct process with annotation events that are similar to GenSpec 6. The assertions can not be failures divergent checked ('[FD-]') because the events that represent the low level signals which have been hidden, determine the initial state and thus the annotation that occurs.
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0,1}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
  chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC3 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]

PROC3 = PROC_STORAGECOMPONENT_ANNOTATE_OUTER(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  < (chan6, chan7, <chan8,chan9> )
>{},
  <1, 2 >,
  < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
>{},
  <3, 4 >
)

alpha_PROC6 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]

PROC6 = PROC_STORAGECOMPONENT_DESIRED GENERIC_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4,chan5> ),
  (chan6, chan7, <chan8,chan9> )
>{},
  < (chan10, chan11, <chan12,chan13> ),
  (chan14, chan15, <chan16,chan17> )
>{}
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC6 \ {internalChoice} )
GEN_SPEC3 = PROC3
GEN_SPEC5 = PROC6
GEN_SPEC6 = PROC STORAGECOMPONENT_HIGHER OUTER SPEC(2, <1, 2>, <3, 4>)

-- GenSpec1 with annotation events added by GenSpec 5 running in parallel
GEN_SPEC1 WITHANNOTATIONS = ( GEN_SPEC1 \ [ alpha_PROC3 ] \ GEN_SPEC5 )
-- The annotations only that were added to GenSpec1 by GenSpec 5
GEN_SPEC1ANNOTATIONS ONLY = ( GEN_SPEC1 WITHANNOTATIONS \ alpha_PROC6 )

-- Check that GenSpec4 with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC1ANNOTATIONS ONLY T= GEN_SPEC6
assert GEN_SPEC6 T= GEN_SPEC1ANNOTATIONS ONLY
```

Figure 106: Example of GenSpec Assertion 11 for Data Storage Specification
Appendix F: Multi-Type Component Generic Specification Model Example

F.2.12 GenSpec Assertions 12: GenSpec 5 Does Not Introduce Extra Behaviour

These assertions (see Figure 107) demonstrates that GenSpec 5 does not add extra behaviours, but only adds events that conceptually annotates the process it is run in parallel with. This is shown by the fact that the process run in parallel with GenSpec 5, with the annotations then hidden is equivalent to the initial process by itself.
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
READALLOWED.allowedIDS | DONTREAD.allowedIDS |
DATABASE.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
  chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC3 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]
PROC3 = PROC_STORAGECOMPONENT_ANNOTATE_OUTER(
  chan0,
  chan1,
  chan2, chan3, <chan4,chan5>,
  chan6, chan7, <chan8,chan9>,
  chan10, chan11, <chan12,chan13>,
  chan14, chan15, <chan16,chan17>
)

alpha_PROC6 = [ chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]
PROC6 = PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC(
  chan0,
  chan1,
  chan2, chan3, <chan4,chan5>,
  chan6, chan7, <chan8,chan9>,
  chan10, chan11, <chan12,chan13>,
  chan14, chan15, <chan16,chan17>
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC6 \ {internalChoice} )
GEN_SPEC5 = PROC1

-- GenSpec1 with annotation events added by GenSpec5 running in parallel
GEN_SPEC1_WITH_ANNOTATIONS = ( GEN_SPEC1 \ [ alpha_PROC3 ] ) GEN_SPEC5
-- The annotations only that were added to GenSpec1 by GenSpec5
GEN_SPEC1_HIDDEN_ANNOTATIONS = (GEN_SPEC1_WITH_ANNOTATIONS \ [ | annotation | ] )

-- Check that GenSpec4 with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC1_HIDDEN_ANNOTATIONS [FD= GEN_SPEC1
assert GEN_SPEC1 [FD= GEN_SPEC1_HIDDEN_ANNOTATIONS

Figure 107: Example of GenSpec Assertions 12 for Data Storage Specification
```
F.2.13  GenSpec Assertions 13: GenSpec 4 With Signals Hidden Is Similar to GenSpec 6

These assertions (see Figure 108) demonstrates that GenSpec 4 correctly driven with the events that represent the low level signals hidden, performs in a similar manner to GenSpec 6. The processes can not be failures divergent checked ("*PD=") both ways against each other, as the driving signal events (which are hidden) determine the initial annotation state that occurs. The hidden events become tau e events preceding the initial annotations, which GenSpec 6 does not contain.
Appendix F: Multi-Type Component Generic Specification Model Example

```plaintext
-- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
DATABIT.allowedIDS.{0..2}.{0, 1}

channel annotation: STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10, chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC1 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 }

PROC1 = PROC_STORAGECOMPONENT_ANNOTATED_SPEC(
  chan0,
  chan1,
  < (chan2, chan3, <chan4, chan5> ), (chan6, chan7, <chan8, chan9> ) >,
  <1, 2 >,
  < (chan10, chan11, <chan12, chan13> ), (chan14, chan15, <chan16, chan17> ) >,
  <3, 4 >
)

alpha_PROC4 = { chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11, chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14, chan3, chan6 }

PROC4 = PROC_STORAGECOMPONENT_CTRL(
  chan0,
  chan1,
  < (chan2, chan3, <chan4, chan5> ), (chan6, chan7, <chan8, chan9> ) >,
  < (chan10, chan11, <chan12, chan13> ), (chan14, chan15, <chan16, chan17> ) >
)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC4
GEN_SPEC4 = ( PROC1 \ {internalChoice} )
GEN_SPEC6 = PROC_STORAGECOMPONENT_HIGHER_OUTER_SPEC(2, <1, 2>, <3, 4> )

-- GenSpec4 limited by the control specification GenSpec3
GEN_SPEC4_WITH_CONTROL = ( GEN_SPEC4 \ alpha_PROC4 ) GEN_SPEC3

-- GenSpec4 limited by the control specification GenSpec3, annotation only
GEN_SPEC4_ANNOTATIONS_ONLY = ( GEN_SPEC4_WITH_CONTROL \ alpha_PROC1 )

-- Check that GenSpec4 with hidden signals is similar to GenSpec6
-- These can not be failures divergent checked 'FD' both ways, as the low
-- level driving signals that proceeds the corresponding annotation events and
-- determine their occurrences, are hidden.
assert GEN_SPEC4_ANNOTATIONS_ONLY [FD= GEN_SPEC6
assert GEN_SPEC6 [T= GEN_SPEC4_ANNOTATIONS_ONLY

Figure 108: Example of GenSpec Assertions 13 for Data Storage Specification

F.3 Similarities to Single Type Component Specifications

Apart from the models and assertions covered in section F.1 and F.2, similarities between this multi type generic component specification and single type generic specifications that represent its interfaces (both the different types of interfaces it can provide, along with its
Appendix F: Multi-Type Component Generic Specification Model Example

ability to provide multiple instances of those types). The specification for the multi type component has the models of the single type interface specifications incorporated directly into its self, this is because this component is concerned with the interactions between instances of these interfaces, and as such the main aim of the following assertions is to provide confidence that this is achieved.

Figure 109: Assertions Demonstrating Similarities with Single Type Generic Interface Specifications

```
--- channel declarations
allowedIDS = {0..4}
datatype STATES =
  START.allowedIDS | FINISH.allowedIDS | NOTFINISHED.allowedIDS |
  RESET.allowedIDS | ERROR.allowedIDS | IDLE.allowedIDS | STORE.allowedIDS |
  STORED.allowedIDS | NOTSTORED.allowedIDS | READ.allowedIDS |
  READALLOWED.allowedIDS | DONTREAD.allowedIDS |
  DATABIT.allowedIDS.{0..2}.{0, 1}
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3, chan4, chan5, chan6, chan7, chan8, chan9, chan10,
  chan11, chan12, chan13, chan14, chan15, chan16, chan17 : {0, 1}

--- Create an instance of the models to check
--- The alpha_PROC contains the low level channels used by the processes
alpha_PROC4 = [chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]
PROC4 = PROC_STORAGECOMPONENT_CONTROL(
  chan0,
  chan1,
  <(chan2, chan3, <chan4, chan5> ),
  (chan6, chan7, <chan8, chan9> )
  >,
  <(chan10, chan11, <chan12, chan13> ),
  (chan14, chan15, <chan16, chan17> )
)

alpha_PROC5 = [chan8, chan0, chan12, chan13, chan15, chan1, chan17, chan11,
  chan2, chan4, chan9, chan5, chan7, chan10, chan16, chan14,
  chan3, chan6 ]
PROC5 = PROC_STORAGECOMPONENT_GENERIC_SPEC(
  chan0,
  chan1,
  <(chan2, chan3, <chan4, chan5> ),
  (chan6, chan7, <chan8, chan9> )
  >,
  <(chan10, chan11, <chan12, chan13> ),
  (chan14, chan15, <chan16, chan17> )
  )

--- The single type interface components
alpha_INTERFACE_1 = [chan0, chan1, chan2, chan3, chan4, chan5 ]
INTERFACE_1 =
  PROC_DATASTORE_GENERIC_SPEC(chan0, chan1, chan2, chan3, <chan4, chan5> )
INTERFACE_1_WITH_CONTROLL =
  ( INTERFACE_1
    [| alpha_INTERFACE_1 |]
  )
  PROC_DATASTORE_CONTROL(chan0, chan1, chan2, chan3, <chan4, chan5> )
```

213
Appendix F: Multi-Type Component Generic Specification Model Example

alpha_INTERFACE_2 = [{ chan0, chan1, chan6, chan7, chan8, chan9 }]
INTERFACE_2 =
  PROC_DATASTORE_GENERIC_SPEC(chan0, chan1, chan6, chan7, <chan8,chan9>)
INTERFACE_2_WITH_CONTROLL =
  ( INTERFACE_2
    [ [ alpha_INTERFACE_2 ] ]
    PROC_DATASTORE_CONTROLL(chan0, chan1, chan6, chan7, <chan8,chan9>)
  )

alpha_INTERFACE_3 = [{ chan0, chan1, chan10, chan11, chan12, chan13 }]
INTERFACE_3 =
  PROC_DATASTORE_GENERIC_SPEC(chan0, chan1, chan10, chan11, <chan12,chan13>)
INTERFACE_3_WITH_CONTROLL =
  ( INTERFACE_3
    [ [ alpha_INTERFACE_3 ] ]
    PROC_DATASTORE_CONTROLL(chan0, chan1, chan10, chan11, <chan12,chan13>)
  )

alpha_INTERFACE_4 = [{ chan0, chan1, chan14, chan15, chan16, chan17 }]
INTERFACE_4 =
  PROC_DATASTORE_GENERIC_SPEC(chan0, chan1, chan14, chan15, <chan16,chan17>)
INTERFACE_4_WITH_CONTROLL =
  ( INTERFACE_4
    [ [ alpha_INTERFACE_4 ] ]
    PROC_DATASTORE_CONTROLL(chan0, chan1, chan14, chan15, <chan16,chan17>)
  )

-- The collection of single type interface components
MULTI_INTERFACES_WITH_CONTROLL =
  < (INTERFACE_1_WITH_CONTROLL, alpha_INTERFACE_1),
    (INTERFACE_2_WITH_CONTROLL, alpha_INTERFACE_2),
    (INTERFACE_3_WITH_CONTROLL, alpha_INTERFACE_3),
    (INTERFACE_4_WITH_CONTROLL, alpha_INTERFACE_4)>

COMBINE_INTERFACES(clock, reset, interfaces) =
  let
    A = reset? _ -> clock.1 -> A
    B( < (xp, xa), (yp, ya) > ^ z ) =
      length(z) == 0 & ( xp ( inter(xa, ya) ] yp )
      [ ]
      length(z) > 0 &
      B( < ((xp) inter(xa, ya) ] yp ), union(xa, ya) > ^ z )
  within B( < (A, [ clock, reset ]) > ^ z )

-- Models
GEN_SPEC2 = ( PROC5 \ {internalChoice} )
GEN_SPEC3 = PROC4
GEN_SPEC2_WITH_CONTROLL = GEN_SPEC2 [ ] GEN_PROC4 [] GEN_SPEC3
INTERFACES = COMBINE_INTERFACES(chan0, chan1, MULTI_INTERFACES_WITH_CONTROLL)

-- Prove that the correctly driven multi type generic specification is a
-- refinement of multiple correctly driven single type components run in
-- parallel
assert INTERFACES [ T= GEN_SPEC2_WITH_CONTROLL

-- Prove that the correctly controlled single type interface components
-- behaviour is covered in the behaviour of the multi type generic component
-- This is done for each of the instances of the interfaces
assert (GEN_SPEC2_WITH_CONTROLL \ diff(alpha_PROC5, alpha_INTERFACE_1) )
  [ T= INTERFACE_1_WITH_CONTROLL
assert (GEN_SPEC2_WITH_CONTROLL \ diff(alpha_PROC5, alpha_INTERFACE_2) )
  [ T= INTERFACE_2_WITH_CONTROLL
assert (GEN_SPEC2_WITH_CONTROLL \ diff(alpha_PROC5, alpha_INTERFACE_3) )
  [ T= INTERFACE_3_WITH_CONTROLL

214
Appendix F: Multi-Type Component Generic Specification Model Example

\begin{verbatim}
assert (GEN_SPEC2_WITH_CONTROLL \ diff(alpha_PROC5, alpha_INTERFACE_4) )
(T= INTERFACE_4_WITH_CONTROLL
\end{verbatim}

F.4 Conclusion & Evaluation

The combination of the assertions covered in section F.2 links various properties of the various models covered in section F.1. This builds up confidence with the models specified so that implemented components can both utilise them in their own proofs and be checked against them. It provides processes so that a low hardware level model of the components can be refinement checked against, a higher clock cycle based conceptual specification to refinement check against, and a method to link the two types of models together to show consistency between them. The assertions covered in section F.3 demonstrate that the multi type generic component specification is a refinement of a combination of single type generic specifications, this enables parts of this components outer boundary to be utilised and substituted where the corresponding generic single type specifications were utilised.
Appendix G  Multi-Type Component
Implementated Model Example

This section will cover and explain the CSP models required for an implemented component, along with the assertions that need to be checked to link the models to each other, thus demonstrating that an implemented component performs as required and within the behaviours dictated by its generic super-type component.

G.1 Models & Specifications

The ‘internalChoice’ event that may appear within the code examples has been utilised instead of internal choice (i.e. ‘\(|-|\)’) to enable ‘chase’ compression to applied if desired. The ‘internalChoice’ event must be hidden for the specifications to be valid, but if ‘chase’ compression has been chosen, the event should only be hidden after ‘chase’ has been applied, otherwise the specification becomes invalid.

G.1.1 ImpSpec 1: Valid Low Level Behaviour

This CSP model (see Figure 110) which is similar to the model defined in section F.1.1, specifies all the valid and allowable low level behaviour that this implemented component may perform at its outer boundary. It may utilises internal choice to determine the possible output behaviour it can perform, although it is not a requirement (e.g. boolean true, boolean false, SKIP, STOP, all have well defined fixed behaviours that do not rely on other internal components). It is useful to note that some implemented components may have the allowable interface boundary behaviour that is identical to that of its generic super type component (e.g. boolean comparisons, PAR), where as other components will have an interface boundary behaviour that is a refinement of its super type component (e.g. boolean true, boolean false, SEQ).

Figure 110: Low Level 'FlipFlopStorage' Component Desired Specification

| channel chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC: {0..2} |
| channel chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC |
| channel chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC: {0..4} |
Appendix G: Multi-Type Component Implemented Model Example

channel chan_readbits_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC: {0..2}. {0, 1}
channel chan_storebits_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC: {0..2}. {0, 1}

PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC(clock, reset, stores, reads) =
  let
  A =
    length(stores) == 0 and length(reads) == 0 & E
      []
    length(stores) != 0 and length(reads) == 0 & B
      []
    length(stores) == 0 and length(reads) != 0 & C
      []
    length(stores) != 0 and length(reads) != 0 & D
  B =
    let
    BA =
      []
      union(
        [] reset,
        clock,
        chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
        
        chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
        y y<-{3,4}
      )
      x:set(stores) @ BB(x)
    ]
    reset.1,
    clock,
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC,
    chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
      )
    )
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC ->
    chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC

BB( (store, stored, data) ) =
  let
  BBA =
    []
    let
    clock,
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC,
    chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
      )
    )
    chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC

BBB =
  []
chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
  x:set(data) @ x?0 -> SKIP
[]
chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
  x:set(data) @
  ( x?0 -> SKIP
  []
  x?1 -> SKIP
  )

)
Appendix G: Multi-Type Component Implemented Model Example

```plaintext

;  
  
  clock?1 ->
   chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC ->
   BBB
  )

  BBC =
   reset?1 ->
   {
   store?0 -> chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> SKIP
   []
   store?1 -> chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.1 -> SKIP
   )
   ;
   BBD
   
   []
   reset?0 ->

   store?0 -> chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> BBD
   []
   store?1 ->
   chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
   chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
   BBE
   )
   []
   store?0 ->
   chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
   ( ( reset?0 -> SKIP
   []
   reset?1 -> SKIP
   )
   ;
   BBD
   
   []
   store?1 ->
   chan_mid_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
   ( reset?0 ->
   chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
   BBE
   []
   reset?1 -> BBD
   )
   BBD =
   clock?1 ->
   ( ( chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
   SKIP
   []
   chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
   SKIP
   )
   ;
   (stored?0 ->
   chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC ->
   BBE
   )
   BBE =
   clock?1 ->
   ( chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
   stored?1 ->
   chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC ->
   BBE
   []
```

218
Appendix G: Multi-Type Component Implemented Model Example

```
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
  stored!0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
BBF
)
BBF =
reset?1 ->
store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
BBD
[]
reset?0 ->
store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
BBG
[]
store?0 -> chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
  ( reset?1 -> BBG
  []
  reset?0 -> BBG
)
BBG =
clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
  stored!0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
BBF
within stored!0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
BBG
   within BA
   let
   CA =
   union(
   { reset,
   clock,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC
   })
   { chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.y
   | y<-{3,4} }
)
[] x:set(reads) @ CB(x)
]
[] {[ reset,1,
clock,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC
]}
[]
( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
P )
[] ([ chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC
]
) CB( (read, readallowed, data) ) =
let
CBA =
reset?1 ->
  ( ( read?0 -> SKIP
  []
  read?1 -> SKIP
  )
);```
Appendix G: Multi-Type Component Implemented Model Example

```
CBB
()
[
reset?0 ->
  ( read?0 -> CBB
  []
  read?1 -> CBC
  )
  []
read?0 ->
  ( ( reset?0 -> SKIP
    []
    reset?1 -> SKIP
    )
  );
CBB
[]
read?1 ->
  ( reset?0 -> CBC
  []
  reset?1 -> CBB
  )
CBB =
clock?1 ->
  ( channel_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
    SKIP
  []
  channel_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
    SKIP
  );
channel_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> CBA
)
CBC =
clock?1 ->
  ( channel_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
    readallowed11 ->
    ( ( readallowed11 x: set(data) o x!0 -> SKIP )
  );
channel_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> CBA
)
[]
channel_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
readallowed10 ->
( ( readallowed10 x: set(data) o x!0 -> SKIP )
  );
channel_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> CBD
)
)
CBD =
reset?1 -> read?0 -> CBB
[]
reset?0 -> read?0 -> CBE
[]
read?0 ->
  ( reset?1 -> CBB
  []
```
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
reset?0 -> CB

CB =
  clock?1 ->
  chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
  readallowed!0 ->
  ( ( ||| x:set(data) & x!0 -> SKIP )
  ;
  chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC -> CBD
  )
  within readallowed!0 ->
  ( ( ||| x:set(data) & x!0 -> SKIP )
  ;
  chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC -> CBA
  )
  within CA
  let
  DA =
  ( ( { [union(
  ( { reset,
  chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
  )
  )
  chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC. y | y\{-3,4\} } )
  || x:set(stores) & DB(x)
  )
  { [union(
  ( { reset,
  clock,
  chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
  )
  )
  chan_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC. y | y\{-3,4\} } )
  || x:set(reads) & DC(x)
  )
  )
  )
  ( chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC,
  chan_readbits_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC,
  chan_storebits_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC,
  clock,
  chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_DESIRED_SPEC
  )
  )
```
Appendix G: Multi-Type Component Implemented Model Example

```
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC

DB( (store, stored, data) ) =
let
  DBA =
  ( DBB
  | chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
  clock
  |
  | chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC
  |)
  |
  DBC =
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
  ( | | x.set(data) @ x?0 -> SKIP
  )
; chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
  ( DBBB(data)
  ;
  DBBA
  )

DBBB(x) =
  length(x) == 0 & STOP
[]
  length(x) == 1 & DBBC(head(x), length(data) - 1)
[]
  length(x) > 1 &
  ( DBBC(head(x), length(data) - length(x))
  |
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.2,
  clock.1
  |
  ]
  DBBB(tail(x))
)

DBBC(x, y) =
( ( x?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.2 ->
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.y.0 ->
clock?1 ->
  SKIP
[]
clock?1 -> SKIP
)
  |
  x?1 ->
)
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.2 ->
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.y.1 ->
clock?1 ->
  SKIP
[]
```

222
Appendix G: Multi-Type Component Implemented Model Example

clock?1 -> SKIP

; 1
(chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> SKIP)

;<
within DBBA
DBC =
reset?1 ->
{

store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
SKIP
[

store?1 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
SKIP
;
;
DBD
)
[
reset?0 ->
{

store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
DBD
[

store?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.2 ->
DBE
)
[

reset?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
{
{
reset?0 -> SKIP
[

reset?1 -> SKIP
;
;
DBD
)
[

reset?1 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->

reset?0 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.2 ->
DBE
[

reset?1 -> DBD
)

DBD =
clock?1 ->
{
{
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
SKIP
[
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
SKIP
;
;
}

stored:0 ->
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DBC
)
)
DBE =
clock?1 ->
(chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
stored?1 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DBC[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
stored?0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DBF[]
)
DBF =
reset?1 ->
store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
DBD[
] reset?0 ->
store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
DBG[
] store?0 ->
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 ->
(reset?1 -> DBD[
][
] reset?0 -> DBG)
)
DBG =
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
stored?0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DBF within{
stored?0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DBA
DC( (read, readallowed, data) ) =
let
DCA =
reset?1 ->
( ( read?0 -> SKIP[]
       read?1 -> SKIP)
   ;
   DCB)
[]
reset?0 ->
( ( read?0 -> DCB[]
       read?1 ->
       chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
       DCC)
[]
read?0 ->
( ( reset?0 -> SKIP[]

```
Appendix G: Multi-Type Component Implemented Model Example

```
reset?1 -> SKIP

DCB

[]
read?1 ->
  (reset?0 ->
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 ->
    DCC
    reset?1 -> DCB
  )

DCB =
clock?1 ->
  ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
    SKIP
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
    SKIP
  );
  (readallowed?0 ->
    ( ||| x : set(data) @ x!0 -> SKIP )
  );
  
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> DCA
)

DCC =
  ( DCD[

| chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
  readallowed,
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
  reset,
  read,
  clock
|

  |]
DCD =
  let
    DCDA =
      clock?1 ->
      chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
      readallowed?0 ->
      chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
      DCD
[ ]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
  readallowed?1 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
  SKIP
  )

DCDB =
reset?1 ->
read?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed?0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
```

225
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
clock? 1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
readallowed! 0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DCDB

within DCDA
DCE =
   let
     DCEA (x) =
       length (x) == 0 & STOP
     []
       length (x) == 1 &
     chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.
       (length(data) - length(x))? z ->
       DCEB (head(x), z)
       []
       length (x) > 1 &
       ( ( chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.
           (length(data) - length(x))? z ->
           DCEB (head(x), z)
           )
       )
     []
     chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
     readallowed,
     chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC,
     reset,
     read,
     clock
   )
   DCEA (tail(x))

DCEB (x, y) =
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
readallowed! 0 ->
x!0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DCEC (x)

[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
( readallowed! 0 ->
x!0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DCED (x, y)
) []
readallowed! 1 ->
x!y ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP

DCEC (x) =

reset? 1 ->
reset? 0 ->
read? 0 -> DCDC
[]
read? 0 ->
( reset? 1 ->
clock? 1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed! 0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP
[]
reset? 0 -> DCDC
)

DCDC =
```

226
read?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP
[
]
reset?0 ->
read?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DCEC(x)
[
]
read?0 ->

( reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP
[
]
reset?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
DCEC(x)
)

DCED(x, y) =
reset?1 ->
read?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP
[
]
reset?0 -> read?0 -> clock?1 -> DCEB(x, y)
[
]
read?0 ->

( reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
readallowed?0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
SKIP
[
]
reset?0 -> clock?1 -> DCEB(x, y)
)

within clock?1 -> DCEA(data)
within {
readallowed?0 ->
( ( ||| x:set(data) & x:0 -> SKIP )
; ( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC -> DCA )
)
within DA
-- No Reads or Stores
E =
Appendix G: Multi-Type Component Implemented Model Example

```
( reset?0 -> SKIP
    [])
reset?1 -> SKIP
); ( clock?1 -> E )
F =
let
FA =
reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FA
[]
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FA
[
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> FB
]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 -> FC
FB =
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FA
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> FD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 -> FC
PC =
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FA
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> FD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 -> FC
FD =
reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FA
[]
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.4 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC ->
FD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.0 -> FD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.1 -> FD
within FA
G =
let
GA =
length(stores) != 0 & GB(head(stores))
[]
length(reads) != 0 & GB(head(reads))
GB((_, _, data)) =
[ [ reset.1 ] ] x:{0..(length(data)-1)} @ GC(x, 0)
GC(x, y) =
reset?1 -> GC(x, 0)
[]
chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC.x:y ->
GC(x, y)
```
Appendix G: Multi-Type Component Implemented Model Example

G.1.2 ImpSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 111) is similar to the model covered in section F.1.2, the CSP model is the model covered in section G.1.1 but altered so that it will accept incorrectly driven input signals followed by explicitly defined deadlocking (i.e. STOP).

Figure 111: Low Level 'FlipFlopStorage' Component Generic Specification with Explicit Deadlocking

```plaintext
channel chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC: 0..2
channel chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
channel chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC: {0..4}
channel chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC: {0..2}.{0, 1}
channel chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC: {0..2}.{0, 1}

PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC(clock, reset, stores, reads) =

let
A =
    length(stores) == 0 and length(reads) == 0 &
    length(stores) != 0 and length(reads) == 0 &
    length(stores) == 0 and length(reads) != 0 &
    length(stores) != 0 and length(reads) != 0 &
B =
    let
        BA =
            ( ( [ ]
                union(
                    ( [ ]
                        reset,
                        clock,
                        chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
                    )
                    ,
                    ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y
                        | y<-{3,4}
                    )
                )
                | x:set(stores) @ BB(x)
            )
            ( [ ]
                reset.1,
                clock,
                chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
                chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
            )
        )
    )
```

229
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->

BB( (store, stored, data) ) =
let
  BBA =
  ( BBB
    []
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
  )
  chann_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC

  BBB =
  ( chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    ( [ ]
      x?0 -> SKIP
      []
      x?1 -> STOP
    )
  )
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
  ( [ ]
    x?0 -> SKIP
    []
    x?1 -> SKIP
  )
  ]

  clock?1 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  BBB

  BBC =
  reset?1 ->
  ( []
    store?0 -> chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 -> SKIP
    []
    store?1 -> chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 -> SKIP
  )

  []
  BBD

  reset?0 ->
  ( []
    store?0 -> chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 -> BBD
    []
    store?1 -> chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 ->
```

230
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
BBE
[]
store?0 ->
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  ( ( reset?0 -> SKIP
    []
    reset?1 -> SKIP
  )
  BBD
[]
store?1 ->
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
  ( reset?0 ->
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    BBE
    []
    reset?1 -> BBD
  )
BBD =
clock?1 ->
  ( ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
    SKIP
    []
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
    SKIP
  )
  stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  BBC
)
BBE =
clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERICSPEC.3 ->
  stored?1 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  BBC
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
  stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  BBF
  BBF =
  reset?1 ->
  store?0 ->
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  BBD
[]
store?1 -> STOP
[]
reset?0 ->
[]
reset?0 ->
```

231
Appendix G: Multi-Type Component Implemented Model Example

```
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 -> BBG
  []
  []
  store?1 -> STOP
  []
  store?0
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 ->
  ( reset?1 -> BBD
    []
    reset?0 -> BBG
  )
  []
  store?1 -> STOP
BBG =
  clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.4 ->
  stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  BBF
within stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  BBA
within BA
  C =
  let
    CA =
    ( ( []
        union( |
        | reset,
        | clock,
        | chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC
        |
        )
        |
        chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.y
        | y<-{3,4}
      )
    )
    |
    x:set(reads) @ CB(x)
    |
    { |
      | { |
        | reset.1,
        | clock,
        | chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC,
        | chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC
        |
      |
    )
    ( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
      |
    )
    \ { |
      | chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC,
      | chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC
      |
    )
    CB( (read, readallowed, data) ) =
      let
        CBA =
        reset?1 ->
          ( ( read?0 -> SKIP
            []
            read?1 -> SKIP
          )
          ;
        CBB
```
Appendix G: Multi-Type Component Implemented Model Example

```
reset?0 ->
  ( read?0 -> CBB
    []
    read?1 -> CBC
  )
[]
read?0 ->
  ( ( reset?0 -> SKIP
      []
      reset?1 -> SKIP
    )
    CBB
  )
[]
reset?1 ->
  ( reset?0 -> CBC
    []
    reset?1 -> CBB
  )
CBB =
clock?1 ->
  ( ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
    SKIP
    []
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
    SKIP
  )
    ;
    ( readallowed!0 ->
      ( ||| x:set(data) @ x!0 -> SKIP )
    )
  )
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> CBA
}
)
CBC =
clock?1 ->
  ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
    readallowed!0 ->
      ( ||| x:set(data) @ x!0 -> SKIP )
    ;
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> CBA
  )
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
  readallowed!0 ->
    ( ||| x:set(data) @ x!0 -> SKIP )
    ;
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> CBD
)
}
CBD =
  reset?1 ->
```
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
( read?0 -> CBB
  []
  read?1 -> STOP
  )

( reset?0 ->
  ( read?0 -> CBE
    []
    read?1 -> STOP
    )
  )

( read?0 ->
  ( reset?1 -> CBB
    []
    reset?0 -> CBE
    )
  )

( reset?0 ->
  ( read?1 -> STOP
    CBE =
    clock?1 ->
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
    readallowed!0 ->
    ( ( ||| x: set(data) @ x!0 -> SKIP )
    )
    ( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> CBD
    )
    )
  )

within readallowed!0 ->
( ( ||| x: set(data) @ x!0 -> SKIP )
 ;
 ( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> CBA
  )
 )
within CA
D =
  let
  DA =
    ( ( ( [ union(
    | reset,
    clock,
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
    )
    ,
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y | y<-{3,4} )
    )
    |] x: set(stores) @ DB(x)
    )
    |] union(
    | reset,
    clock,
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC
    )
    ,
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y | y<-{3,4} )
    )
    |] union(
    | reset,
    clock,
```
Appendix G: Multi-Type Component Implemented Model Example

```verbatim
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC

{ chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y | y<>{3,4} }

| x: set(reads) @ DC(x)

} reset.1,

chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
clock,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC

( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  H
)

chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC,
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
channel_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC

DB( (store, stored, data) ) =
let
  DBA =
  ( DBB
  |
  chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
clock
  |
  |
  DBC
  )

chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC

DBB =
let
  DBBA =
( chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  ( x: set(data) @
    ( x?0 -> SKIP
    |
    x?1 -> STOP
    )
  )
)
(clock?1 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DBBA
)
)

chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 ->
( DBB(data)

DBBA

```

235
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
DBBB(x) =
    length(x) == 0 & STOP
    []
    length(x) == 1 & DBBC(head(x), length(data) - 1)
    []
    length(x) > 1 &
      ( DBBC(head(x), length(data) - length(x))
      []
    }

    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.2,
    clock.1

    []
    DDBB(tail(x))
    }

DBBC(x, y) =
    ( ( x? 0 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.2 ->
    chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y.0 ->
    clock?1 ->
    SKIP
    []
    clock?1 -> SKIP
    )
    []
    x?1 ->
    )

chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.2 ->
chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.y.1 ->
clock?1 ->
SKIP
[]
clock?1 -> SKIP
}

chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> SKIP
)

) within DDBA
DDB =
    reset?1 ->
      ( store?0 ->
      chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    SKIP
    []
    store?1 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
    SKIP
      )
      ;
    DDB
    )
[
    reset?0 ->
    ( store?0 ->
```
Appendix G: Multi-Type Component Implemented Model Example

```
chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
DBD
[]
store?1 ->
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.2 ->
    DBE
)
[]
store?0 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    ( ( reset?0 -> SKIP
       []
       reset?1 -> SKIP
    );
    DBD
    )
[]
store?1 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
    (reset?0 ->
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
    chan_mid_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.2 ->
    DBE
[]
reset?1 -> DBD
  )
  DBD =
  clock?1 ->
  ( ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
      SKIP
      []
      chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
      SKIP
    );
  )
  (stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DBC
  )
  )
  DBE =
  clock?1 ->
  ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
    stored?1 ->
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
    DBC
  []
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
  stored?0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DBF
  )
  DBF =
  reset?1 ->
  )
```
store?0 ->
  chan_mid_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  DBD
[
  ]
store?1 -> STOP
[
  ]
reset?0 ->
[
  ]
reset?1 -> STOP
[
  ]
reset?0 -> chan_mid_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 -> DBG
[
  ]
store?0 ->
  chan_mid_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  (reset?1 -> DBG
  [
  ]
reset?0 -> DBG
[
  ]
store?1 ->
  chan_mid_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  (reset?1 -> DBG
  [
  ]
reset?0 -> DBG
[
  ]
reset?0 ->
  chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.0 ->
  stored0 ->
  chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DBF
within (
  stored0 ->
  chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DBA
  )
  DC( (read, readallowed, data) ) =
  let
  DCA =
  reset?1 ->
  ( (read?0 -> SKIP
    [
    ]
  read?1 -> SKIP
  );
  DCB
  );
  reset?0 ->
  ( (read?0 -> DCB
    [
    ]
  read?1 ->
  chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
  DCC
  );
  reset?0 ->
  ( (reset?0 -> SKIP
    [
    ]
  reset?1 -> SKIP
  );
  DCB
  )
Appendix G: Multi-Type Component Implemented Model Example

[]
read?1 ->
  (reset?0 ->
      chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.1 ->
      DCC
take
    reset?1 -> DCB)

DCB =
clock?1 ->
  (chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
    SKIP
take
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
    SKIP)

; (readallowed?0 ->
  (|| x:set(data) @ x:0 -> SKIP ))
)
;
(chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC -> DCA)
)

DCE =
  (DCD
take
)

DCD =
  let
    DCDA =
      clock?1 ->
        (chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
          readallowed?0 ->
          chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
          DCDB
take
        chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
          readallowed?1 ->
          chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
          SKIP
        DCDB =
          reset?1 ->
            (read?0 ->
             clock?1 ->

239
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
readallowed!0 ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
SKIP

[]
read?1 -> STOP

[]
reset?0 ->
  (read?0 -> DCDC
   []
   read?1 -> STOP
  )
[]
read?0 ->

reset?1 ->
clock?1 ->
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
readallowed!0 ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
SKIP
[]
reset?0 -> DCDC

)[]
read?1 -> STOP

DCDC =
clock?1 ->
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
readallowed!0 ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
DCDB

within DCDA
DCE =
let
  DCEA(x) =
    length(x) == 0 & STOP
[]
    length(x) == 1 &
    chan_readbits_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.
    (length(data) - length(x))? z ->
    DCEB(head(x), z)
[]
    length(x) > 1 &
    ( (chan_readbits_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC.
    (length(data) - length(x))? z ->
    DCEB(head(x), z)
  )[]
    chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
readallowed,
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_GENERIC_SPEC,
reset,
read,
clock
[])

|]

DCEA(tail(x))

)
Appendix G: Multi-Type Component Implemented Model Example

```
DCEC(x, y) =
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
  readallowed!0 ->
  x!0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  DCEC(x)
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
  ( readallowed!0 ->
    x!0 ->
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
    DCEC(x, y)
  []
  readallowed!1 ->
    x!1 y! ->
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
    SKIP
)

DCEC(x) =
  reset?1 ->
    ( read?
      clock?1 ->
      chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
      readallowed!0 ->
      x!0 ->
      chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
      SKIP
    []
    read?1 -> STOP
    )
  []
  reset?0 ->
    ( read?
      clock?1 ->
      chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
      readallowed!0 ->
      x!0 ->
      chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
      DCEC(x)
    []
    read?1 -> STOP
    )
  []
reset?1 ->
  clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
  readallowed!0 ->
  x!0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  SKIP
[]
reset?0 ->
  clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.4 ->
  readallowed!0 ->
  x!0 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
```

241
Appendix G: Multi-Type Component Implemented Model Example

```
DCBC(x)
)

[]
read?1 -> STOP

DCED(x, y) =
reset?1 ->

read?0 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC.3 ->
readallowed:0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC ->
SKIP

[]
read?1 -> STOP

[]
reset?0 ->
( read?0 -> clock?1 -> DCB(x, y)
[]
read?1 -> STOP
)
[]
read?0 ->

reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC.3 ->
readallowed:0 ->
x:0 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC ->
SKIP

[]
reset?0 -> clock?1 -> DCB(x, y)

[]
read?1 -> STOP
within clock?1 -> DCB(data)
within ( readallowed:0 ->
( ( | | | x: set(data) @ x:0 -> SKIP )
;
( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC -> DCA )
)

within DA
-- No Reads or Stores
E =
( reset?0 -> SKIP
[]
reset?1 -> SKIP
); ( clock?1 -> E )
F =
let
FA =
reset?1 ->
clock?1 ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC.3 ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC SPEC ->
FA
```

242
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
[1] 
clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC.3 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_GENERIC_SPEC ->
  FA
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 -> FB
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 -> FC
FB =
clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.3 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  FA
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 -> FD
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 -> FD
FC =
clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.3 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  FA
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 -> FD
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 -> FC
FD =
reset?1 ->
  clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.3 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  FA
[
] 
reset?1 ->
  clock?1 ->
  chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.4 ->
  chan_link_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC ->
  FD
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.0 -> FD
[
] 
chan_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.1 -> FD
within FA
G =
let
  GA =
    length(stores) != 0 & GB(head(stores))
  [ ]
  length(reads) != 0 & GB(head(reads))
  GB( (_ , _ , data) ) =
    [ ]
  reset.1 ] [ ]
  x:{0..(length(data)-1)} @ GC(x, 0)
  GC(x, y) =
  reset?1 -> GC(x, 0)
  [ ]
  chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.x?y ->
    GC(x, y)
  [ ]
  chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT GENERIC_SPEC.x?z ->
    GC(x, z)
  within GA
H =
```

243
Appendix G: Multi-Type Component Implemented Model Example

G.1.3 ImpSpec 3: Model of Implemented Logic

This CSP model (see Figure 112), is a model of the segment of logic that this component represents and is achieved through modelling the individual logic components and running them in parallel.

Figure 112: CSP Model of the Logic Circuit Segment of the 'FlipFlopStorage' Component
Appendix G: Multi-Type Component Implemented Model Example

alpha_PROC29 = { chan11, chan13, chan50 }
alpha_PROC29 = PROC_AND(chan13, chan11, chan50)
alpha_PROC30 = { chan15, chan50, chan17 }
alpha_PROC30 = PROC_AND(chan17, chan15, chan50)
alpha_PROC31 = { chan21, chan7, chan23 }
alpha_PROC31 = PROC_AND(chan7, chan23, chan21)
alpha_PROC32 = { chan47, chan6, chan29, chan2 }
alpha_PROC32 = PROC_XOR(chan29, chan47, chan2, chan6)
alpha_PROC33 = { chan18, chan20 }
alpha_PROC33 = PROC_GND(chan47)
alpha_PROC34 = { chan19, chan35, chan50 }
alpha_PROC34 = PROC_AND(chan19, chan35, chan50)
alpha_PROC35 = { chan47 }
alpha_PROC35 = PROC_GND(chan47)
alpha_PROC36 = { chan15, chan16, chan22 }
alpha_PROC36 = PROC_AND(chan16, chan15, chan22)
alpha_PROC37 = { chan15, chan43, chan25 }
alpha_PROC37 = PROC_AND(chan15, chan25, chan43)
alpha_PROC38 = { chan36, chan48, chan38 }
alpha_PROC38 = PROC_AND(chan36, chan48, chan38)
alpha_PROC39 = { chan8, chan41, chan4, chan39 }
alpha_PROC39 = PROC_OR(chan41, chan39, chan4, chan8)
alpha_PROC40 = { chan42, chan3, chan23 }
alpha_PROC40 = PROC_AND(chan3, chan23, chan42)
alpha_PROC41 = { chan35, chan33 }
alpha_PROC41 = PROC_GND(chan35, chan33)
alpha_PROC42 = { chan46, chan0, chan1, chan25 }
alpha_PROC42 = PROC_DTYPE(chan0, chan1, chan46, chan25)
alpha_PROC43 = { chan0, chan20, chan26, chan1 }
alpha_PROC43 = PROC_DTYPE(chan0, chan1, chan20, chan26)
alpha_PROC44 = { chan44, chan6, chan38, chan2 }
alpha_PROC44 = PROC_OR(chan38, chan44, chan2, chan6)
alpha_PROC45 = { chan24, chan34, chan9, chan5 }
alpha_PROC45 = PROC_OR(chan24, chan34, chan9, chan5)

-- The outer level signals for the component being tested
SYSTEM_INTERFACE =
{ chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 }

-- The alphabet of signals used in the model of the logic
SYSTEM_ALPHA =
{ chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25, chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9, chan31, chan12, chan32, chan33, chan34, chan17, chan14, chan35, chan15, chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42, chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4, chan1, chan49, chan0, chan50, chan8, chan5 }

-- System Declaration of the internal logic components and the signals they use
SYSTEM_LIST =
< PROC5, alpha_PROC5 >,
< PROC6, alpha_PROC6 >,
< PROC7, alpha_PROC7 >,
< PROC8, alpha_PROC8 >,
< PROC9, alpha_PROC9 >,
< PROC10, alpha_PROC10 >,
< PROC11, alpha_PROC11 >,
< PROC12, alpha_PROC12 >,
< PROC13, alpha_PROC13 >,
< PROC14, alpha_PROC14 >,
< PROC15, alpha_PROC15 >,
< PROC16, alpha_PROC16 >,
< PROC17, alpha_PROC17 >,
< PROC18, alpha_PROC18 >,
< PROC19, alpha_PROC19 >,
< PROC20, alpha_PROC20 >,
Appendix G: Multi-Type Component Implemented Model Example

```
(PROC21, alpha_PROC21 ),
(PROC22, alpha_PROC22 ),
(PROC23, alpha_PROC23 ),
(PROC24, alpha_PROC24 ),
(PROC25, alpha_PROC25 ),
(PROC26, alpha_PROC26 ),
(PROC27, alpha_PROC27 ),
(PROC28, alpha_PROC28 ),
(PROC29, alpha_PROC29 ),
(PROC30, alpha_PROC30 ),
(PROC31, alpha_PROC31 ),
(PROC32, alpha_PROC32 ),
(PROC33, alpha_PROC33 ),
(PROC34, alpha_PROC34 ),
(PROC35, alpha_PROC35 ),
(PROC36, alpha_PROC36 ),
(PROC37, alpha_PROC37 ),
(PROC38, alpha_PROC38 ),
(PROC39, alpha_PROC39 ),
(PROC40, alpha_PROC40 ),
(PROC41, alpha_PROC41 ),
(PROC42, alpha_PROC42 ),
(PROC43, alpha_PROC43 ),
(PROC44, alpha_PROC44 ),
(PROC45, alpha_PROC45 )
```

-- The logic model of the implemented component. 'ImpSpec3'
SYSTEM =
  ( REPL(SYSTEM_LIST) \ diff(SYSTEM_ALPHA, SYSTEM_INTERFACE))
)
\ {internalChoice}

-- Used to run the logic components in parallel
REPL(p) =
  let
  INNER1(pl, a1, p2, a2) = p1 [ inter(a1, a2) |] p2
  INNER2( </(p1,a1)>"<(p2,a2)>"p3 ) =
    null(p3) & INNER1(pl,a1,p2,a2)
  []
  not null(p3) & INNER2( </(INNER1(p1,a1,p2,a2), union(a1,a2)>"p3 )
  INNER3( </(p1,-)>) = p1
  within ( null(p) & STOP
  []
  length(p) == 1 & INNER3(p)
  []
  length(p) > 1 & INNER2(p) )

G.1.4 ImpSpec 4: Annotation Only Specification

This CSP model (see Figure 113), is the expected behaviour of the 'FlipFlopStorage' component from an annotation only perspective, with annotation id values for each of the interfaces along with the bit width of the storage having to be supplied. The CSP model will demonstrate all the possible behaviours that the 'FlipFlopStorage' component can perform, thus describing how driving the components interfaces interacts and affect each other.
Appendix G: Multi-Type Component Implemented Model Example

**Figure 113: FlipFlopStorage Component Annotation Only Specification**

```
channel chan_mid_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC: {0..2}
channel chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC
channel chan_readbits_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC: {0..2}. {0, 1}
channel chan_storebits_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC: {0..2}. {0, 1}
PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC(bitLength, sid, rid)

let
A =
length(sid) == 0 and length(rid) == 0 & SKIP
[]
length(sid) != 0 and length(rid) == 0 & B
[]
length(sid) == 0 and length(rid) != 0 & C
[]
length(sid) != 0 and length(rid) != 0 & D

B =
let
BA =
( [ | { ý} {ý} annotation.RESET, chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC
|]
|] x: set(sid) * BB(x)
)
[|] chan_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
annotation.RESET
|
]
BD
BB(x) =
( ( | | y: set(sid) * annotation.RESET.y -> SKIP ); BB(x) )
[|]
annotation.IDLE.x ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
BB(x)
[|]
annotation.STORE.x ->
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 ->
( ( | | y:{0...(bitLength-1)} ) * ( annotation.DATABIT.x.y.0 -> SKIP
[]
annotation.DATABIT.x.y.1 -> SKIP )
);
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
( chan_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC.4 ->
annotation.NOTSTORED.x ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
BC(x)
[]
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
annotation.STORED.x ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
BB(x)
)
)
)
BC(x) =
( ( | | y: set(sid) * annotation.RESET.y -> SKIP ); BB(x) )
[]
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
chan_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC.4 ->
annotation.NOTSTORED.x ->
chan_link_PROC_FLIPPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
BC(x)

BD =
```
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
( ( ||| y: set(sid) @ annotation.RESET. y -> SKIP ) ; BD )
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 -> BD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 -> BE
BE =
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 -> BD
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 -> BF
BF =
( ( ||| y: set(sid) @ annotation.RESET. y -> SKIP ) ; BD )
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.4 -> BF
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 -> BF
within BA \ { chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC }

C =
let
CA =
[] || annotation.RESET,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
[]
[] x: set(rid) @ CB(x)
CB(x) =
( ( ||| y: set(rid) @ annotation.RESET. y -> SKIP ) ; CB(x) )
[]
annotation.IDLE.x ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
[]
annotation.READ.x ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
[]
annotation.READALLOWED.x ->
( ( ||| y: {0..(bitLength-1)} @ annotation.DATABIT.x.y.0 -> SKIP ) ;
( chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
) )
within CA \ { chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC }

D =
let
DA =
( ( ||
|| annotation.RESET,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
||
{ chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
} )
[] x: set(sid) @ DB(x)
[]
union{!! annotation.RESET,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
||
{ chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
} )
[]
( { union({! annotation.RESET,
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
||
{ chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.
} )
} )
```

248
Appendix G: Multi-Type Component Implemented Model Example

```plaintext

I Y <- \{2, 3\}
x: set(rid) ® DD(x)

I ý ý ý ci

{ \{ chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 annotation.RESET
 \}

Iý {ý chan_link_PROC_FLIPFLOPSTORAGECOMPONSNP_HIGHER SPEC,
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC,
 chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC

-- Store
DB(x) =
 chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
 ( ( \{ || y:union(set(sid), set(rid)) \} ® annotation.RESET.y -> SKIP )
 
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 ->
 annotation.IDLE.x ->
 chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.1 ->
 ( ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.2 -> SKIP
 []
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 -> SKIP
 )
 
 annotation.STORE.x ->
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0 ->
 ( ( || y:0..(bitLength-1)) ®
   annotation.DATABIT.x.0.y.0 ->
   chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0.y.0 ->
   SKIP
 []
 annotation.DATABIT.x.0.y.1 ->
 chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.0.y.1 ->
   SKIP
 
 annotation.DATABIT.x.1.y.0 ->
 chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.1.y.0 ->
   SKIP
 []
 annotation.DATABIT.x.1.y.1 ->
 chan_storebits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.1.y.1 ->
   SKIP
 
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.2 ->
 annotation.STORED.x ->
 DB(x)
 []
 chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
 annotation.NOTSTORED.x ->
 DC(x)
 )
 )

 DC(x) =
 chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
 ( ( || y:union(set(sid), set(rid)) \} ® annotation.RESET.y -> SKIP )
)```

249
Appendix G: Multi-Type Component Implemented Model Example

```
; DB (x)
)
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
annotation.NOTSTORED.x ->
DC(x)
)
-- Reads
DD(x) =
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
( ( ( [ ] [ ] y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
; DD(x)
)
)

annotation.IDLE.x ->
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
( ( chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.2 -> SKIP
[]
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 -> SKIP
)
; DD(x)
)
[]

annotation.READ.x ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.1 ->
( ( ( [ ] [ ] union{
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.y,
    annotation.READALLOWED.x,
    annotation.DONTREAD.x,
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
| y<-{2, 3}
}

[ [ annotation.RESET ] ]
)
[ ] y:{0..(bitLength-1)} @
chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.y.0 -> DG(x, y, 0)
[]
chan_readbits_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.y.1 -> DG(x, y, 1)
)
[]

[ [ union{
    chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.y,
    annotation.READALLOWED.x,
    annotation.DONTREAD.x,
    chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC
| y<-{2, 3}
}

[ [ annotation.RESET ] ]
)
[ ]
DF(x)
)
; DD(x)
)

DE(x) =
( ( [ ] [ ] y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
[]
chan_link_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC ->
chan_PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
```

250
Appendix G: Multi-Type Component Implemented Model Example

```
annotation.DONTREAD.x ->
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
DE(x)

DF(x) =
  ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
[]
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
  ( chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.2 ->
    annotation.READALLOWED.x ->
    SKIP
  []
    annotation.DONTREAD.x ->
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
    annotation.DONTREAD.x ->
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
    DE(x)
  )
]
DG(x, y, z) =
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
  ( chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.2 ->
    []
      annotation.READALLOWED.x ->
      annotation.DATABIT.x.y.z -> SKIP
    []
      annotation.DONTREAD.x ->
      DH(x, y, z)
    []
    annotation.DONTREAD.x ->
    chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
    annotation.DONTREAD.x ->
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
    DI(x)
  )
]
DH(x, y, z) =
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
  ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
[]
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
  ( chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.2 ->
    []
      annotation.READALLOWED.x ->
      annotation.DATABIT.x.y.z -> SKIP
    []
      annotation.DONTREAD.x ->
      DH(x, y, z)
    []
    annotation.DONTREAD.x ->
    chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
    annotation.DONTREAD.x ->
    chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
    DI(x)
  )
]
DI(x) =
  ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
[]
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
chan_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.3 ->
annotation.DONTREAD.x ->
chan_link_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC ->
DI(x)
)
)-- Storage
DJ =
  ( [[ || annotation.RESET ]] ] ] x:[0..(bitLength-1)] @ DK(x, 0) )
[ | [ || annotation.RESET ]] ] ]
DL
DK(x, y) =
  ( ||| y:union(set(sid), set(rid)) @ annotation.RESET.y -> SKIP )
    ;
    DK(x, 0)
  []
chan_readbits_PROC_FLIPPLOPSTORAGECOMPONENT_HIGHER_SPEC.x.y ->
```

251
Appendix G: Multi-Type Component Implemented Model Example

G.2 Assertions: Linking the Models Together

G.2.1 ImpSpec Assertion 1: Deadlock-Free

The assertion stated in Figure 114 checks that the model of the segment of logic circuit is deadlock-free. This check demonstrates two properties, the first is that there exists no
loops consisting of only clocked or non-clocked logic components, the second is that any internal components are guaranteed to be driven correctly so long as the outer component is driven correctly. The guarantee that internal components are driven correctly is possible because the models of the internal components are models that accept all possible inputs (both valid and incorrect), with the incorrect inputs being followed by explicitly defined deadlock i.e. 'STOP' (see section D.1.2). If the STOP's are not reached, then invalid inputs to internal components have not been created or propagated through. The demonstrated multi type implemented component does not utilise other internal components, this check is here to provide the functionality for enabling the development of other multi type components which may require this feature.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan22, chan23, chan24, chan25, chan26, chan27, chan28, chan29, chan30, chan31, chan32, chan33, chan34, chan35, chan36, chan37, chan38, chan39, chan40, chan41, chan42, chan43, chan44, chan45, chan46, chan47, chan48, chan49, chan50, chan51, chan52, chan53, chan54, chan55, chan56 : {0, 1}
-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = [{chan15, chan11, chan12, chan16, chan9, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9}
PRO6 = PROC_STORAGECOMPONENT_CONTROLL(
chan0, chan1,
< (chan2, chan3, <chan4,chan5> ) ,
(chan6, chan7, <chan8,chan9> )
>
, < (chan10, chan11, <chan12,chan13> ),
(chan14, chan15, <chan16,chan17> )
>
)-- An implementation of the logic to check
SYSTEM_INTERFACE =

{ chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 }

IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PRO6

-- Deadlock-free check the expected correct generic component
assert (IMP_SPEC3 || SYSTEM_INTERFACE || GEN_SPEC3) : [deadlock free [P]]
```

Figure 114: 'FlipFlopStorage' Component Deadlock-Free Assertion
G.2.2 ImpSpec Assertion 2: Super Type Control Only Limits the Behaviour

The assertion stated in Figure 115 demonstrates that the process that dictates allowable correct driving input signals of the super type of this component, does only limit the behaviour of this implemented component, thus ensuring that it does not introduce any new behaviour.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25, chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9, chan31, chan12, chan32, chan33, chan17, chan14, chan35, chan15, chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42, chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4, chan1, chan49, chan50, chan8, chan5 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = {[| chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 |

PROC6 = PROC_STORAGECOMPONENT_CONTROLL(
    chan0, chan1,
    < (chan2, chan3, <chan4, chan5> ),
    <chan6, chan7, <chan8, chan9> >
)

< (chan10, chan11, <chan12, chan13> ),
( chan14, chan15, <chan16, chan17> )
)

-- An implementation of the logic to check
SYSTEM_INTERFACE =
  {[| chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 |

IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

-- Check that the control specification only limits the behaviour of the segment
-- of logic, and does not introduce new behaviour
assert IMP_SPEC3 [T= (IMP_SPEC3 [| SYSTEM_INTERFACE |] GEN_SPEC3)]
```

Figure 115: Super Type Component Limits the Behaviour of the Implementation

G.2.3 ImpSpec Assertion 3: Expected Component Behaviour is Deadlock-Free

The assertion stated in Figure 116 demonstrates that the expected boundary behaviour that the model of the implementation of the logic circuit segment will be checked against
is deadlock-free. This is to provide better confidence in this model's correctness as it will be used in future checks.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25,
chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9,
chan31, chan12, chan32, chan33, chan34, chan17, chan14, chan35, chan15,
chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42,
chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4,
chan1, chan49, chan50, chan8, chan5 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC{
  chan0, chan1,
  < {chan2, chan3, <chan4,chan5> } ,
  < {chan6, chan7, <chan8,chan9> } 
> ,
  < {chan10, chan11, <chan12,chan13> } ,
  < {chan14, chan15, <chan16,chan17> } 
> }

assert IMP_SPEC1 : [deadlock free [F]]
```

Figure 116: Expected Component Behaviour is Deadlock-Free

**G.2.4 ImpSpec Assertion 4: Expected Boundary Behaviour Refines Super Type**

The assertion stated in Figure 117 demonstrates that the expected correct boundary behaviour of the implemented component is a valid refinement of the expected boundary behaviour of its generic super type component.
Appendix G: Multi-Type Component Implemented Model Example

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25,
chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9,
chan31, chan12, chan32, chan33, chan34, chan17, chan14, chan35, chan15,
chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42,
chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4,
chan1, chan49, chan50, chan8, chan5 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_FLIPIFLOPSTORAGECOMPONENT_DESIRED_SPEC(
    chan0, chan1,
    < (chan2, chan3, <chan4,chan5> ) ,
    (chan6, chan7, <chan8,chan9> )
>
, (<chan10, chan11, <chan12,chan13> ),
    (chan14, chan15, <chan16,chan17> )
>
)
GEN_SPEC1 =
( PROC_STORAGECOMPONENT_DESIRED_GENERIC_SPEC(
    chan0, chan1,
    < (chan2, chan3, <chan4,chan5> ) ,
    (chan6, chan7, <chan8,chan9> )
>
, < (chan10, chan11, <chan12,chan13> ),
    (chan14, chan15, <chan16,chan17> )
>
)
\ {internalChoice}
)
assert GEN_SPEC1 [T= IMP_SPEC1

Figure 117: Expected Component Behaviour is a Refinement of Super Type

G.2.5 ImpSpec Assertions 5: Correctly Driven Implementation Behaves as Expected

The assertions stated in Figure 118 demonstrate that the segment of logic circuit for this implemented component, if driven correctly, behaves as expected.
Appendix G: Multi-Type Component Implemented Model Example

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25,
chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9,
chan31, chan12, chan32, chan33, chan14, chan15, chan35, chan15,
chan36, chan17, chan38, chan39, chan5, chan40, chan41, chan3, chan42,
chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4,
chan1, chan49, chan50, chan8, chan5 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC(
chan0, chan1,
< (chan2, chan3, <chan4,chan5> ) ,
(chan6, chan7, <chan8,chan9> )
>
,
< (chan10, chan11, <chan12,chan13> ) ,
(chan14, chan15, <chan16,chan17> )
>
)

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = {
| chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5,
chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4,
chan7, chan9 |
}
PROC6 = PROC_STORAGECOMPONENT_CONTROL(
chan0, chan1,
< (chan2, chan3, <chan4,chan5> ) ,
(chan6, chan7, <chan8,chan9> )
>
,
< (chan10, chan11, <chan12,chan13> ) ,
(chan14, chan15, <chan16,chan17> )
>
)

-- An implementation of the logic to check
SYSTEM_INTERFACE =
{| chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan2,
chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 |
}
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6
assert IMP_SPEC1 [FD=
((IMP_SPEC3 [] SYSTEM_INTERFACE []) GEN_SPEC3) \ {\ annotations }]
assert ((IMP_SPEC3 [[ SYSTEM_INTERFACE ]] GEN_SPEC3)
\ {\ annotations }
) [FD= IMP_SPEC1

Figure 118: Correctly Driven Component Behaves as Expected

G.2.6 ImpSpec Assertion 6: Annotation Outer Level Does Not Introduce Deadlock

The assertion stated in Figure 119 demonstrates that annotating the outer level of the implemented component with the annotation process specified by its super type (i.e. GenSpec 5, see Section F.1.5), does not introduce any deadlocks.
Appendix G: Multi-Type Component Implemented Model Example

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25, chan7, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9, chan31, chan12, chan32, chan33, chan34, chan17, chan14, chan35, chan15, chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42, chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4, chan1, chan49, chan50, chan8, chan5 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC(
    chan0, chan1, 
    < (chan2, chan3, <chan4,chan5> ), (chan6, chan7, <chan8,chan9> ) >, 
    < (chan10, chan11, <chan12,chan13> ) ), (chan14, chan15, <chan16,chan17> ) > 
)

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = [ 
    chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9
]

PROC6 = PROC_STORAGECOMPONENT_CONTROL(
    chan0, chan1, 
    < (chan2, chan3, <chan4,chan5> ), (chan6, chan7, <chan8,chan9> ) >, 
    < (chan10, chan11, <chan12,chan13> ) ), (chan14, chan15, <chan16,chan17> ) >
)

alpha_PROC5 = [ 
    chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9
]

PROC5 = PROC_STORAGECOMPONENT_ANNOTATE_OUTER(
    chan0, chan1, 
    < (chan2, chan3, <chan4,chan5> ) >, 
    (chan6, chan7, <chan8,chan9> ) >, 
    <1, 3 > 
    < (chan10, chan11, <chan12,chan13> ) >, 
    (chan14, chan15, <chan16,chan17> ) > 
)

-- An implementation of the logic to check
SYSTEM_INTERFACE = 
    [ chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9 ]
)

IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6
GEN_SPEC5 = PROC5

assert ( (IMP_SPEC3 [[ SYSTEM_INTERFACE ]] GEN_SPEC3) 
    [[ SYSTEM_INTERFACE ]] GEN_SPEC5 ) : [deadlock free [F]]
```

Figure 119: Annotating outer level of the component does not introduce deadlock
G.2.7 ImpSpec Assertion 7: Expected High Level Behaviour is Deadlock-Free

The assertion stated in Figure 120 demonstrates that the high level model describing the expected behaviour of the implemented component is deadlock-free. This helps to build confidence in the model for when using it in future checks.

```
-- Create an instance of the models to check
IMP_SPEC4 = PROC_FLIPFLOPSTORAGECOMPONENT_HIGHER_SPEC(2, <1, 3>, <5, 7>)
assert IMP_SPEC4 : [deadlock free[P]]
```

Figure 120: Components High Level Behaviour is Deadlock-Free

G.2.8 ImpSpec Assertions 8: Component Behaves Similarly to Expected Higher Spec

The assertions stated in Figure 121 demonstrate that the annotations obtained from the implemented segment of logic circuit, performs in a similar manner to that of the expected higher behavioural specification. The test can not be failure divergence checked both ways (one has to be a trace refinement), this is due to the way annotations are added to the outer layer. As the outer level input annotations occur after the corresponding low level input signal events (i.e. the events that represent the wires), hiding these low level signal events causes the high level model extracted from the implemented segment of logic circuit to appear to have internal choice determining the high level conceptual input states. The internal choice for the inputs does not really exist, but appears because the events that do determine what occurs though external choice have been hidden (i.e. the low level signals). Through altering the process of annotating the outer level of a component (see Appendix H), it is possible to simplify the extracted model so that it directly equivalent to the expected higher behaviour.
Appendix G: Multi-Type Component Implemented Model Example

--- channel declarations
channel internalChoice
channel chan0 : [1]
channel chan18, chan19, chan20, chan21, chan13, chan22, chan23, chan24, chan25, chan27, chan26, chan27, chan10, chan28, chan29, chan2, chan30, chan9, chan31, chan12, chan32, chan33, chan34, chan17, chan14, chan35, chan15, chan36, chan37, chan38, chan39, chan6, chan40, chan41, chan3, chan42, chan43, chan44, chan11, chan45, chan46, chan47, chan16, chan48, chan4, chan1, chan49, chan50, chan8, chan5 : [0,1]

--- Create an instance of the models to check
IMP_SPEC1 = PROC_FLIPFLOPSTORAGECOMPONENT_DESIRED_SPEC(
  chan0, chan1,
  < (chan2, chan3, <chan4,chan5> ), (chan6, chan7, <chan8,chan9> ) >,
  < (chan10, chan11, <chan12,chan13> ), (chan14, chan15, <chan16,chan17> ) >
)

--- Create an instance of the models to check
--- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = { chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9
  }
PROC6 = PROC_STORAGECOMPONENT_CONTROL(
  chan0, chan1,
  < (chan2, chan3, <chan4,chan5> ), (chan6, chan7, <chan8,chan9> ) >,
  < (chan10, chan11, <chan12,chan13> ), (chan14, chan15, <chan16,chan17> ) >
)

alpha_PROC5 = { chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9
  }
PROC5 = PROC_STORAGECOMPONENT_ANNOTATE OUTER(
  chan0, chan1,
  < (chan2, chan3, <chan4,chan5> ), (chan6, chan7, <chan8,chan9> ) >,
  <1, 3 >,
  < (chan10, chan11, <chan12,chan13> ), (chan14, chan15, <chan16,chan17> ) >,
  <5, 7 >
)

--- An implementation of the logic to check
SYSTEM_INTERFACE =
  { chan15, chan11, chan12, chan16, chan0, chan6, chan10, chan5, chan17, chan2, chan8, chan3, chan14, chan13, chan1, chan4, chan7, chan9
  }
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6
GEN_SPEC5 = PROC5

assert (( (IMP_SPEC3 [ | SYSTEM_INTERFACE | ] GEN_SPEC3)
  [ | SYSTEM_INTERFACE | ] GEN_SPEC5 ) \ SYSTEM INTERFACE
  ) [P= Imp_Model_3]
assert Imp_Model_3 [T= (( (IMP_SPEC3 [ | SYSTEM_INTERFACE | ] GEN_SPEC3)
  [ | SYSTEM_INTERFACE | ] GEN_SPEC5 ) \ SYSTEM_INTERFACE )

Figure 121: Component Behaves Similarly to Expected Higher Behaviour
G.3 Conclusions & Evaluation

The combination of the assertions covered in section G.2 links various properties of the various models covered in section G.1 together and also with some of the models covered in section F.1 (i.e. the models for its corresponding super type). This builds up confidence with the implemented component through crosschecking various properties hold true throughout the various models created for it, along with the implementation being a refinement of its super type component, thus allowing the implemented component to be placed wherever its super type has been used.

G.4 Future Work

G.4.1 Linking Clock Cycle Annotations to Higher Specification

Similar to the work described in section D.4.1, the CSP model covered in section G.1.4 could be linked to a higher conceptual description of the component that gives the sequencing of the required conceptual events at a software level, and not a hardware clock cycle level.
Appendix H  Alternative Generic Specification Model Example

This section will cover and explain the simplified CSP models required for a generic super-type component, along with the assertions that need to be checked to link the models to each other. The main modification arises from the process that controls the driving of a component supplying the driving signals, instead of just limiting what signals can be accepted. This modification also affects the method of annotating the outer layer of a component, which will have the effect of simplifying the higher level clock based annotation specification of implemented components, thus causing the annotation of the start signals preceding the start signals of internal components, and not being specified as occurring in parallel which the current implementation indicates. The main benefit of this is that when the annotations are simplified to the software based CSP model, it will more directly mimic the application source code and becomes more apparent when proving an implemented component. The adaptation of the processes also causes some minor modifications of the assertions that are required for the proof to hold true, along with the introduction of a ‘NOTRESET’ annotation.

H.1  Models & Specifications

The ‘internalChoice’ event that may appear within the code examples has been utilised instead of internal choice (i.e. ‘|1|’) to enable ‘chase’ compression to applied if desired. The ‘internalChoice’ event must be hidden for the specifications to be valid, but if ‘chase’ compression has been chosen, the event should only be hidden after ‘chase’ has been applied, otherwise the specification becomes invalid.

H.1.1  GenSpec 1: Valid Low Level Behaviour

This model (see Figure 122) specifies all the valid and allowable low level behaviour of this type of super type component. The purpose is to describe the interface boundary behaviours, thus enabling implemented components to refinement check against it,
proving there behaviours are within the requirements for it to be a sub-type of this supertype.

```
PROC_PROCESS_DESIRED_GENERIC_SPEC(clock, reset, start, finish) =
  let
    A =
      start?x -> reset?y -> C(x, y) []
      reset?y -> start?x -> C(x, y)
    B =
      start?0 -> reset?y -> D(y) []
      reset?y -> start?0 -> D(y)
    C(x, y) =
      y == 1 & clock?1 -> finish!0 -> A []
      y == 0 &
        ( x == 0 & clock?1 -> finish!0 -> A []
        x == 1 & clock?1 ->
          ( internalChoice -> finish!1 -> A [] -- |-
            internalChoice -> finish!0 -> B )
        )
    D(y) =
      y == 1 & clock?1 -> finish!0 -> A []
      y == 0 & clock?1 ->
        ( internalChoice -> finish!1 -> A [] -- |-
          internalChoice -> finish!0 -> B )
  within finish!0 -> A
```

Figure 122: Low Level Generic Control Flow Specification

This specification will only accept correct input driving signals, and will return valid output result signals. Internal choice is utilised to enable it to specify all the possible valid refinements.

H.1.2 GenSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 123) is based on the model covered in section H.1.1, but altered to also accepts invalid driving input signals to be submitted to it. These invalid input driving signals are followed by an explicitly defined 'STOP', that will explicitly deadlock the model should it ever be reached. Similar to the specification in section H.1.1, the returned output signals will be all the valid possible permutations allowed (internal choice is utilised to create those permutations, so long as it is driven correctly). The reason why this model will accept invalid driving signals is to enable the
demonstration that the driving control process will only provide correct valid driving signals.

PROC_PROCESS_GENERIC_SPEC(clock, reset, start, finish) =
let
A =
  start?x -> reset?y -> C(x, y)
  []
reset?y -> start?x -> C(x, y)
B =
  start?1 -> STOP
  []
start?0 -> reset?y -> D(y)
  []
reset?y ->
  ( start?0 -> D(y)
    []
    start?1 -> STOP
  )
C(x, y) =
  y == 1 & clock?1 -> finish!0 -> A
  []
y == 0 &
  ( x == 0 & clock?1 -> finish!0 -> A
    []
x == 1 & clock?1 ->
    ( internalChoice -> finish!1 -> A
      [] -- |-
      internalChoice -> finish!0 -> B
    )
  )
D(y) =
  y == 1 & clock?1 -> finish!0 -> A
  []
y == 0 & clock?1 ->
  ( internalChoice -> finish!1 -> A
    [] -- |-
    internalChoice -> finish!0 -> B
  )
within finish!0 -> A

Figure 123: Low Level Generic Control Flow Specification with Explicit Deadlocking

H.1.3 GenSpec 3: Correct Component Driving

This CSP model (see Figure 124) is used to provide the valid input driving signals to an implemented sub-type component while testing is performed. It uses internal choice to achieve this, and has external choice to enable both correct and incorrect output signals to occur. The invalid output signals are followed by an explicitly defined deadlock ‘STOP’, which combined with deadlock-free checks will enabled the component being tested to be proven to only output valid signals if it is correctly driven.
PROC_PROCESS_DRIVER(clock, reset, start, finish) =
let
  A =
  start ! 1 ->
  ( reset ! 1 -> D
  | -|
  reset ! 0 -> C
  )
  | -|
  start ! 0 ->
  ( reset ! 1 -> D
  | -|
  reset ! 0 -> D
  )
  | -|
  reset ! 1 ->
  ( start ! 1 -> D
  | -|
  start ! 0 -> D
  )
  | -|
  reset ! 0 ->
  ( start ! 1 -> C
  | -|
  start ! 0 -> D
  )
  B =
  start ! 0 ->
  ( reset ! 1 -> D
  | -|
  reset ! 0 -> C
  )
  | -|
  reset ! 1 -> start ! 0 -> D
  | -|
  reset ! 0 -> start ! 0 -> C
  C =
  clock?1 ->
  ( finish?0 -> B
  []
  finish?1 -> A
  )
  D =
  clock?1 ->
  ( finish?0 -> A
  []
  finish?1 -> STOP
  )
within finish?0 -> A

Figure 124: Generic Control Flow Specification - Correct Signal Driver

H.1.4 GenSpec 4: Annotated Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 125) is derived from the one covered in section D.1.2, but containing extra events added to describe conceptually what is occurring. The aim of this is to enable a link between a low level hardware model and a higher level conceptual meaning of the function the hardware is performing. The added 'id' parameter added to
the process is to provide a method to distinguish between different instances of this process. The annotation events depicting the states that are entered into from how this component is driven can only be specified after the event has occurred, where as the output signals are controlled by this component and so the corresponding annotation events can be performed before outputting the signals. The reason why renaming can not be used to obtain a higher level conceptual model of what is occurring, thus the required use of extra events depicting the annotations, is because the same signal states can mean different things depending on the state of the system (e.g. a start signal high state 'start?1' can mean that this component has been triggered, or that this component is being driven incorrectly and an error has occurred).

\[
\begin{align*}
\text{channel} & \text{ chan\_contro11\_flow\_AnnotatedSpec : } \{1, 2\}, \{0, 1\} \\
\text{PROC\_PROCESS\_ANNOTATED\_SPEC}(\text{clock, reset, start, finish, id}) &= \\
\text{let A = } \\
\text{start?x} & \rightarrow \text{reset?y} \rightarrow C(x, y) \\
\text{[} & \\
\text{reset?y} & \rightarrow \text{start?x} \rightarrow C(x, y) \\
\text{B = } & \\
\text{start?1} & \rightarrow \text{annotation.ERROR.id} \rightarrow \text{STOP} \\
\text{[} & \\
\text{start?0} & \rightarrow \text{reset?y} \rightarrow D(y) \\
\text{[} & \\
\text{reset?y} & \rightarrow \\
\text{start?0} & \rightarrow D(y) \\
\text{[} & \\
\text{start?1} & \rightarrow \text{annotation.ERROR.id} \rightarrow \text{STOP} \\
\text{C(x, y) = } & \\
\text{y == 1} & \& \text{annotation.RESET.id} \rightarrow \text{clock?1} \rightarrow \text{finish!0} \rightarrow A \\
\text{[} & \\
\text{y == 0} & \& \\
\text{x == 0} & \& \text{annotation.IDLE.id} \rightarrow \text{clock?1} \rightarrow \text{finish!0} \rightarrow A \\
\text{[} & \\
\text{x == 1} & \& \text{annotation.START.id} \rightarrow \text{clock?1} \rightarrow \\
\left( \text{internalChoice} \rightarrow \text{annotation.FINISH.id} \rightarrow \text{finish!1} \rightarrow A \\
\text{[} & \\
\text{internalChoice} & \rightarrow \text{annotation.NOTFINISHED.id} \rightarrow \text{finish!0} \rightarrow B \\
\text{D(y) = } & \\
\text{y == 1} & \& \text{annotation.RESET.id} \rightarrow \text{clock?1} \rightarrow \text{finish!0} \rightarrow A \\
\text{[} & \\
\text{y == 0} & \& \text{annotation.NOTRESET.id} \rightarrow \text{clock?1} \rightarrow \\
\left( \text{internalChoice} \rightarrow \text{annotation.FINISH.id} \rightarrow \text{finish!1} \rightarrow A \\
\text{[} & \\
\text{internalChoice} & \rightarrow \text{annotation.NOTFINISHED.id} \rightarrow \text{finish!0} \rightarrow B \\
\text{within finish!0} & \rightarrow A
\end{align*}
\]

Figure 125: Annotated Generic Control Flow Specification with Explicit Deadlocking
H.1.5 GenSpec 5: Correct Component Driving Annotating the Outer Layer

This CSP model (see Figure 126) is used to annotate the outer layer of an implemented sub-type of this component. The process is modified from the specification in section H.1.3, it provides the correct input signals preceded by any relevant annotations, and accepts both correct and incorrect output signals, post fixing any corresponding annotation events that are required after they have occurred.

```plaintext
PROC_PROCESS_ANNOTATED_DRIVER(clock, reset, start, finish) =
let
A =
  annotation.START.0 ->
  ( start !1 -> reset !0 -> C
   [-]
   reset !0 -> start !1 -> C
  )
  [-]
  annotation.IDLE.0 ->
  ( start !0 -> reset !0 -> D
   [-]
   reset !0 -> start !0 -> D
  )
  [-]
  annotation.RESET.0 ->
  ( start !1 -> reset !1 -> D
   [-]
   start !0 -> reset !1 -> D
   [-]
   reset !1 ->
   ( start !1 -> D
     [-]
     start !0 -> D
   )
  )
B =
  annotation.RESET.0 ->
  ( start !0 -> reset !1 -> D
   [-]
   reset !1 -> start !0 -> D
  )
  [-]
  annotation.NOTRESET.0 ->
  ( start !0 -> reset !0 -> C
   [-]
   reset !0 -> start !0 -> C
  )
C = clock?1 ->
  ( finish?0 -> annotation.NOTFINISHED.0 -> B
   []
   finish?1 -> annotation.FINISH.0 -> A
  )
D = clock?1 ->
  ( finish?0 -> A
   []
   finish?1 -> annotation.ERROR.0 -> STOP
  )
within finish!0 -> A
```

Figure 126: Generic Control Flow Annotate Outer Layer
Appendix H: Alternative Generic Specification Model Example

H.1.6 GenSpec 6: Clock Cycle Higher Generic Specification

This CSP model (see Figure 127) is an annotation only clock cycle based higher conceptual specification. It is used as a comparison for the extracted annotations from the annotated low level hardware models. The model is sufficiently small so that it is unlikely that 'chase' compression should be needed to be applied, which is why internal choice (i.e. '|-|') is used instead of using an extra event to simulate internal choice.

```plaintext
PROC_PROCESS_HIGHER_OUTER_SPEC(id) =
let
  A =
    annotation.RESET.id -> A
    []
    annotation.START.id -> B
    []
    annotation.IDLE.id -> A
  B =
    annotation.NOTFINISHED.id -> C
    |-
    annotation.FINISH.id -> A
  C =
    annotation.RESET.id -> A
    []
    annotation.NOTRESET.id ->
      ( annotation.NOTFINISHED.id -> C
        |-
        annotation.FINISH.id -> A
      )
within A
```

Figure 127: Generic Control Flow Annotation Specification

H.1.7 GenSpec 7: Driving Clock Cycle Higher Generic Specification

This CSP model (see Figure 128) is the driving component for the annotation only specification GenSpec 6. The combination of these two specifications run in parallel can be used as a basis to perform checks and comparisons against.
Appendix H: Alternative Generic Specification Model Example

PROC PROCESS HIGHER CONTROLL OUTER SPEC (id) let
A =
   annotation. RESET. id -> A
   \-\ annotation. START. id -> B
   \-\ annotation. IDLE. id -> A
B =
   annotation. NOTFINISHED. id -> C
   []
   annotation. FINISH. id -> A
C =
   annotation. RESET. id -> A
   \-\ annotation. NOTRESET. id ->
      ( annotation. NOTFINISHED. id -> C
         []
         annotation. FINISH. id -> A
      )
within A

Figure 128: Control Generic Control Flow Annotation Specification

H.1.8 GenSpec 8: Annotating Clock Cycle Higher Generic Specification

This CSP model (see Figure 129) is the annotated driving component for the annotation only specification GenSpec 6. The combination of these two specifications run in parallel can be used as a basis to perform checks and comparisons against.

PROC_PROCESS_HIGHER_ANNOTATED_OUTER_SPEC(id) =
let
A =
   annotation. RESET. 0 -> annotation. RESET. id -> A
   \-\ annotation. START. 0 -> annotation. START. id -> B
   \-\ annotation. IDLE. 0 -> annotation. IDLE. id -> A
B =
   annotation. NOTFINISHED. id -> annotation. NOTFINISHED. 0 -> C
   []
   annotation. FINISH. id -> annotation. FINISH. 0 -> A
C =
   annotation. RESET. 0 -> annotation. RESET. id -> A
   \-\ annotation. NOTRESET. 0 -> annotation. NOTRESET. id ->
      ( annotation. NOTFINISHED. id -> annotation. NOTFINISHED. 0 -> C
         []
         annotation. FINISH. id -> annotation. FINISH. 0 -> A
      )
within A

Figure 129: Annotate Generic Control Flow Annotation Specification
H.1.9 GenSpec 9: Software Higher Generic Specification

This model (see Figure 130) provides a software level based model of a generic control flow process. A process when started may or may not ever finish, and it can only be started again if it has finished.

```plaintext
-- internal choice with STOP is provided as an alternative to the FINISH event, this is because conceptually when started, a control flow process does not have to finish and is dependent on itself or other internal components to determine if it does or not.
PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(id) =
let
  A =
    annotation.START.id ->
    ( STOP
      | |
      annotation.FINISH.id -> A
    )
  within A
```

Figure 130: Example of Simplified Control Flow Annotation Specification

H.1.10 GenSpec 10: Annotation Only Control Generic Specification

This model (see Figure 131) provides a process to drive the annotation only specifications using internal choice to select the possible valid driving states.

```plaintext
PROC_PROCESS_CONTROL_ANNOTATED_OUTER_SPEC =
let
  A =
    annotation.RESET.0 -> A
    | |
    annotation.START.0 -> B
    | |
    annotation.IDLE.0 -> A
  B =
    annotation.NOTFINISHED.0 -> C
    |
    annotation.FINISH.0 -> A
  C =
    annotation.RESET.0 -> A
    | |
    annotation.NOTRESET.0 ->
      ( annotation.NOTFINISHED.0 -> C
        |
        annotation.FINISH.0 -> A
      )
  within A
```

Figure 131: Control Process for Annotation Only Control Flow Specification
H.2 Assertions: Linking the Models Together

To ensure consistency between all the models for a generic component, several assertions have to be proven. The consistency between the models is required because the proof of an implemented component can utilise several of these specifications.

H.2.1 GenSpec Assertion 1: Initial Deadlock-Free Check

This initial deadlock-free check (see Figure 132) of GenSpec 1 (see section H.1.1) is to provide a base comparison for future deadlock-free checks and trace refinements.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the model to check
-- The alpha_PROC4 contains the low level channels used by this instance
alpha_PROC4 = { chan0, chan1, chan2, chan3 }
PROC4 = PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Side the internalChoice event to ensure that PROC4 has internal choice
-- performing correctly if needed.
gen_SPEC1 = ( PROC4 \ {internalChoice} )

-- Deadlock-free check the expected correct generic component
assert gen_SPEC1 : [deadlock free [F]]
```

Figure 132: Example of GenSpec Assertion 1 for Controll Flow Process

H.2.2 GenSpec Assertion 2: GenSpec 2 Contains GenSpec 1 Behaviour

This assertion (see Figure 133) demonstrates that GenSpec 2 (see section H.1.2) contains all the behaviour dictated by GenSpec 1 (see section H.1.1), although this assertion allows GenSpec 2 to provide extra behaviours.
Appendix H: Alternative Generic Specification Model Example

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC's contain the low level channels used by the processes
alpha_PROC3 = [[ chan0, chan1, chan2, chan3 ]]
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)
alpha_PROC4 = [[ chan0, chan1, chan2, chan3 ]]
PROC4 = PROC_PROCESS_DESORED GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC2 = ( PROC3 \ {internalChoice} )

-- Check GenSpec 2 contains the behaviour of GenSpec 1
assert GEN_SPEC2 [T= GEN_SPEC1]

Figure 133: Example of GenSpec Assertion 2 for Control Flow Process

H.2.3 GenSpec Assertion 3: GenSpec 3 Compatible with GenSpec 2

This assertion (see Figure 61) demonstrates that the GenSpec 3 controlling specification (see section D.1.3) does not introduce any new behaviour to the specifications it is being run in parallel with. This still leaves the possibility of it limiting the events that can occur, but does not guarantee any properties regarding this.

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC's contain the low level channels used by the processes
alpha_PROC2 = [[ chan0, chan1, chan2, chan3 ]]
PROC2 = PROC_PROCESS DRIVER(chan0, chan1, chan2, chan3)
alpha_PROC3 = [[ chan0, chan1, chan2, chan3 ]]
PROC3 = PROC_PROCESS GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC2 = ( PROC3 \ {internalChoice} )

-- GenSpec2 driven by the controll specification GenSpec3
GEN_SPEC2_DRIVEN = GEN_SPEC2 [[ alpha_PROC2 ]] GEN_SPEC3

-- Check GenSpec2 contains the behaviour of GenSpec2 driven by GenSpec3
assert GEN_SPEC2 [T= GEN_SPEC2_DRIVEN]

Figure 134: Example of GenSpec Assertion 3 for Control Flow Process
H.2.4  GenSpec Assertion 4: GenSpec 3 Removes Deadlock from GenSpec 2

This assertion (see Figure 135) demonstrates that GenSpec 3 (see section H.1.3) does not incorrectly drive the component it is run in parallel with. This does not dictate that GenSpec 3 does not supply all the possible correct driving signals (although it does).

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the model to check
-- She alpha PROC's Contains the low level chars ale used by the procwes
alpha_PROC2 = {{ chan0, chan1, chan2, chan3 }}
PROC2 = PROC_PROCESS_DRIVER(chan0, chan1, chan2, chan3)

alpha_PROC3 = {{ chan0, chan1, chan2, chan3 }}
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = PROC2
GEN_SPEC2 = ( PROC3 \ {internalChoice} )

-- GenSpec2 driven by the controll specification GenSpec3
GEN_SPEC2_DRIVEN = GEN_SPEC2 \ { alpha_PROC2 } GEN_SPEC3

-- Check that GenSpec3 does not invoke the incorrect driving options in GenSpec2
assert GEN_SPEC2_DRIVEN : [deadlock free [F]]
```

Figure 135: Example of GenSpec Assertion 4 for Control Flow Process

H.2.5  GenSpec Assertions 5: GenSpec 3 Removes Only Incorrect Driving

These assertions (see Figure 136) demonstrate that GenSpec 3 drives the process it is run in parallel with, such that it only provides correct driving signals. These assertions also demonstrates that GenSpec 3 does not introduce any extra behaviours and provides all possible correct driving signals, and is achieved through proving that GenSpec 3 run in parallel with GenSpec 2 covers all the traces contained in GenSpec 1. The processes can’t be failure divergent checked ‘[FD-’ both ways, as the driving control process uses internal choice to provide all the possible driving input signals and determine which ones occur, where as the generic specifications (i.e. GenSpec 1) uses external choice to allow the components driving it to dictate what state it should enter (by supplying the corresponding sequence of input signals).
Appendix H: Alternative Generic Specification Model Example

-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha PROC's contains the low level channels used by the processes
alphaPROC2 = { chan0, chan1, chan2, chan3 }
PROC2 = PROC_PROCESS_DRIVER(chan0, chan1, chan2, chan3)

alphaPROC3 = { chan0, chan1, chan2, chan3 }
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

alphaPROC4 = { chan0, chan1, chan2, chan3 }
PROC4 = PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC1 = ( PROC4 \ {internalChoice} )
GEN_SPEC2 = ( PROC3 \ {internalChoice} )
GEN_SPEC3 = PROC2

-- GenSpec2 driven by the control specification GenSpec3
GEN_SPEC2_DRIVEN = GEN_SPEC2 \ {alphaPROC2} \ GEN_SPEC3

-- Check that GenSpec3 in parallel with GenSpec2 is produces all traces covered
-- by GenSpec1
assert GEN_SPEC1 ∧ GEN_SPEC2_DRIVEN
assert GEN_SPEC2_DRIVEN FD GEN_SPEC1

Figure 136: Example of GenSpec Assertions 5 for Control Flow Process

H.2.6 GenSpec Assertions 6: Properties of the Annotation Events

These assertions (see Figure 137) demonstrate that the annotation event contained within GenSpec 4 do not introduce extra behaviours, but are only used to conceptually describe what is occurring. This is achieved through hiding the annotation events, and proving that the resultant process is indistinguishable to the GenSpec 2 model.
Appendix H: Alternative Generic Specification Model Example

Figure 137: Example of GenSpec Assertions 6 for Control Flow Process

H.2.7 GenSpec Assertion 7: GenSpec 3 Compatible with GenSpec 4

This assertion (see Figure 138) proves that the control driving process GenSpec 3 does not add extra behaviours to the annotated generic specification GenSpec 4.

Figure 138: Example of GenSpec Assertion 7 for Control Flow Process
H.2.8 GenSpec Assertion 8: GenSpec 3 Removes Deadlock From GenSpec 4

This assertion (see Figure 139) proves that the control process GenSpec 3 does not drive the annotated generic specification GenSpec 4 into a state that is deadlocked (this also includes the explicitly defined deadlocks 'stop').

```
-- channel declarations
datatype STATES =
  START.(0) | FINISH.(0) | NOTFINISHED.(0) | RESET.(0) | NOTRESET.(0) |
  ERROR.(0) | IDLE.(0)
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC's contains the low level channels used by the processes
alpha_PROC2 = [chan0, chan1, chan2, chan3] |
PROC2 = PROC_PROCESS_DRIVER(chan0, chan1, chan2, chan3)
alpha_PROC0 = [chan0, chan1, chan2, chan3] |
PROC0 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)
-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = (PROC0 \ {internalChoice})
-- GenSpec4 driven by the control specification GenSpec3
GEN_SPEC4_DRIVEN = GEN_SPEC4 \ alpha_PROC2 \ GEN_SPEC3

-- Check that GenSpec4 removes the incorrect driving options from GenSpec4
assert GEN_SPEC4_DRIVEN ::(deadlock free [P])
```

Figure 139: Example of GenSpec Assertion 8 for Control Flow Process

H.2.9 GenSpec Assertions 9: GenSpec 4 is an Annotated GenSpec 2

These assertions (see Figure 140) demonstrate that if the annotation events contained within GenSpec 4 are hidden, then GenSpec is indistinguishable to GenSpec 2.
Appendix H: Alternative Generic Specification Model Example

```
-- channel declarations
datatype STATES =
  START. | FINISH. | NOTFINISHED. | RESET. | NOTRESET. |
  ERROR. | IDLE.
channel annotation : STATES
channel internalChoice
channel chan0 : {1}
channel chan1, chan2, chan3 : {0, 1}
-- Create an instance of the models to check
-- The alpha_PROC's contains the low level channels used by the processes
alpha_PROC3 =|-- chanO, chan1, chan2, chan3 |
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)
alpha_PROC0 =|-- chanO, chan1, chan2, chan3 |
PROC0 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 0)
-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = ( PROC3 \ {internalChoice} )
GEN_SPEC4 = ( PROC0 \ {internalChoice} )
-- GenSpec4 with hidden annotation events
GEN_SPEC4_NO_ANNOTATIONS = ( GEN_SPEC4 \ { annotation | | )
-- Check that GenSpec4 with hidden annotations is indistinguishable to GenSpec2
assert GEN_SPEC4_NO_ANNOTATIONS [FD- GEN_SPEC2
assert GEN_SPEC2 [FD- GEN_SPEC4_NO_ANNOTATIONS
```

Figure 140: Example of GenSpec Assertions 9 for Control Flow Process

H.2.10 GenSpec Assertion 10: GenSpec 6 Deadlock-Free

This assertion (see Figure 141) deadlock-free checks GenSpec 6 which helps to provide a base comparison for future deadlock-free checks and trace refinements for the annotations.

```
-- Channel declarations
datatype STATES =
  START. | FINISH. | NOTFINISHED. | RESET. | NOTRESET. |
  ERROR. | IDLE.
channel annotation : STATES
-- Higher Outer Spec
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)
-- Check that GenSpec6 is deadlock-free
assert GEN_SPEC6 :[deadlock free [F]]
```

Figure 141: Example of GenSpec Assertion 10 for Control Flow Process

H.2.11 GenSpec Assertions 11: GenSpec 5 Annotates Outer Level Correctly

These assertions (see Figure 142) demonstrates that GenSpec 5 will drive a component correctly and annotate the outer level of the process with annotation events that are similar to GenSpec 6.
Appendix H: Alternative Generic Specification Model Example

```plaintext
-- channel declarations
datatype STATES =
  START.(0) | FINISH.(0) | NOTFINISHED.(0) | RESET.(0) | NOTRESET.(0) |
  ERROR.(0) | IDLE.(0)
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Create an instance of the models to check
-- The alpha PROC's contains the low level channels used by the processes
alpha_PROC1 = { chan0, chan1, chan2, chan3 }
PROC1 = PROC_PROCESS_ANNOTATED_DRIVER(chan0, chan1, chan2, chan3)

alpha_PROC3 = { chan0, chan1, chan2, chan3 }
PROC3 = PROC_PROCESS_GENERIC_SPEC(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC2 = ( PROC3 \ {internalChoice} )
GEN_SPEC5 = PROC1
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(0)

-- GenSpec1 with annotation events added by GenSpec 5 running in parallel
GEN_SPEC2_WITH_ANNOTATIONS = ( GEN_SPEC2 \ alpha_PROC1 ) GEN_SPEC5
-- The annotations only that were added to GenSpec1 by GenSpec 5
GEN_SPEC2_ANNOTATIONS_ONLY = (GEN_SPEC2_WITH_ANNOTATIONS \ alpha_PROC3)

-- Check that GenSpec1 controlled correctly and annotated has annotations that
-- are similar to GenSpec6
assert GEN_SPEC2_ANNOTATIONS_ONLY \ T = GEN_SPEC6
assert GEN_SPEC6 \ T = GEN_SPEC2_ANNOTATIONS_ONLY
```

Figure 142: Example of GenSpec Assertion 11 for Control Flow Process

H.2.12 GenSpec Assertions 12: GenSpec 5 Does Not Introduce Unexpected Behaviours

These assertions (see Figure 143) demonstrates that GenSpec 5 does not introduces unexpected behaviours, but only correctly drives a component and adds events that conceptually annotates at the outer layer the process it is run in parallel with. This is shown by the fact that the process run in parallel with GenSpec 5, with the outer annotations then hidden, is equivalent to the component correctly driven by GenSpec 3, and that hiding the low level signals leaves only the expected annotations.
Appendix H: Alternative Generic Specification Model Example

-- channel declarations

datatype STATES =
  START.{0..1} | FINISH.{0..1} | NOTFINISHED.{0..1} | RESET.{0..1} |
  NOTRESET.{0} | ERROR.{0..1} | IDLE.{0..1}

channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

-- Annotation alphabets
alpha_inner =
  { annotation.x.1 | x<-[RESET, NOTRESET, START, IDLE, NOTFINISHED, FINISH] }

alpha_outer =
  { annotation.x.0 | x<-[RESET, NOTRESET, START, IDLE, NOTFINISHED, FINISH] }

-- Create an instance of the models to check
-- The alpha_PROCESS contains the low level channels used by the processes
alpha_PROCESS0 = [{ chan0, chan1, chan2, chan3 }] PROC0 = PROC_PROCESSANNOTATED_SPEC(chan0, chan1, chan2, chan3, 1)

alpha_PROCESS1 = [{ chan0, chan1, chan2, chan3 }] PROC1 = PROC_PROCESSANNOTATED_DRIVER (chan0, chan1, chan2, chan3)

alpha_PROCESS2 = [{ chan0, chan1, chan2, chan3 }] PROC2 = PROC_PROCESS_DRIVER(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = ( PROC0 \ {internalChoice} )
GEN_SPEC5 = PROC1
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(1)
GEN_SPEC7 = PROC_PROCESS_HIGHER_CONTROL_OUTER_SPEC(1)
GEN_SPEC8 = PROC_PROCESS_HIGHERANNOTATED_OUTER_SPEC (1)

-- GenSpec4 driven by GenSpec3
GEN_SPEC4_DRIVEN_BY_3 = ( GEN_SPEC4 ] alpha_PROCESS0 ] GEN_SPEC3 )

-- GenSpec4 driven and annotated by GenSpec5
GEN_SPEC4_DRIVEN_BY_5 = ( GEN_SPEC4 ] alpha_PROCESS0 ] GEN_SPEC5 )
-- GenSpec4 driven and annotated by GenSpec5 with outer annotations hidden
GEN_SPEC4_DRIVEN_BY_5_HIDDEN_OUTER = ( GEN_SPEC4_DRIVEN_BY_5 \ alpha_outer )
-- GenSpec4 driven and annotated by GenSpec5 with low level signals hidden
GEN_SPEC4_DRIVEN_BY_5_HIDDEN_SIGNALS = ( GEN_SPEC4_DRIVEN_BY_5 \ alpha_PROCESS0 )

-- GenSpec6 driven and annotated by GenSpec8
GEN_SPEC6_ANNOTATED = ( GEN_SPEC6 [ alpha_inner ] GEN_SPEC8 )

-- Check that GenSpec4 driven by GenSpec5 with the outer annotations hidden is
-- indistinguishable to GenSpec4 driven by GenSpec3
assert GEN_SPEC4_DRIVEN_BY_5_HIDDEN_OUTER [FD= GEN_SPEC4_DRIVEN_BY_3]
assert GEN_SPEC4_DRIVEN_BY_5 [FD= GEN_SPEC4_DRIVEN_BY_5_HIDDEN_OUTER]

-- Check that GenSpec4 driven by GenSpec5 with the low level signals hidden is
-- indistinguishable to GenSpec6 driven by GenSpec8
assert GEN_SPEC4_DRIVEN_BY_5_HIDDEN_SIGNALS [FD= GEN_SPEC6_ANNOTATED]
assert GEN_SPEC6_ANNOTATED [FD= GEN_SPEC4_DRIVEN_BY_5_HIDDEN_SIGNALS]

Figure 143: Example of GenSpec Assertions 12 for Control Flow Process
H.2.13 GenSpec Assertions 13: GenSpec 4 With Signals Hidden Is Similar to GenSpec 6

These assertions (see Figure 144) demonstrate that GenSpec 4 correctly driven with the events that represent the low level signals hidden, performs in a similar manner to GenSpec 6, and indistinguishable from GenSpec 6 when correctly driven.

--- channel declarations

datatype STATES =
  START.(0..1) | FINISH.(0..1) | NOTFINISHED.(0..1) | RESET.(0..1) |
  NOTRESET.(0) | ERROR.(0..1) | IDLE.(0..1)
channel annotation : STATES

channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan3 : {0, 1}

--- Annotation alphabets
alpha_inner =
  { annotation.x.1 | x<-{RESET, NOTRESET, START, IDLE, NOTFINISHED, FINISH} }
alpha_outer =
  { annotation.x.0 | x<-{RESET, NOTRESET, START, IDLE, NOTFINISHED, FINISH} }

-- Create an instance of the models to check
-- The alpha_PROC's contains the low level channels used by the processes
alpha_PROCO = [{ chan0, chan1, chan2, chan3 }]
PROC0 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan2, chan3, 1)

alpha_PROC2 = [{ chan0, chan1, chan2, chan3 }]
PROC2 = PROC_PROCESS_DRIVER(chan0, chan1, chan2, chan3)

-- Hide the internalChoice event to ensure that processes has internal choice
-- performing correctly if needed.
GEN_SPEC3 = PROC2
GEN_SPEC4 = ( PROC0 \ {internalChoice} )
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(1)
GEN_SPEC7 = PROC_PROCESS_HIGHER_CONTROLL_OUTER_SPEC(1)

-- GenSpec4 driven by GenSpec 3
GEN_SPEC4_DRIVEN_BY_3 = ( GEN_SPEC4 \ [ alpha_PROCO ] \ GEN_SPEC3 )

-- GenSpec4 driven and annotated by GenSpec 5 with low level signals hidden
GEN_SPEC4_DRIVEN_BY_3_HIDDEN_SIGNALS = ( GEN_SPEC4_DRIVEN_BY_3 \ alpha_PROCO )

-- GenSpec6 driven by GenSpec 7
GEN_SPEC6_DRIVEN = ( GEN_SPEC6 \ [ alpha_inner ] \ GEN_SPEC8 )

-- Check that GenSpec driven by GenSpec3 with the low level signals hidden is
-- indistinguishable to GenSpec6 driven by GenSpec7
assert GEN_SPEC4_DRIVEN_BY_3_HIDDEN_SIGNALS [FD= GEN_SPEC6_DRIVEN
assert GEN_SPEC6_DRIVEN [FD= GEN_SPEC4_DRIVEN_BY_3_HIDDEN_SIGNALS

Figure 144: Example of GenSpec Assertions 13 for Control Flow Process
Appendix H: Alternative Generic Specification Model Example

H.2.14 GenSpec Assertions 14: GenSpec 10 Similar to GenSpec 5

The assertions stated in Figure 145 demonstrate the similarity of GenSpec 5 to GenSpec 10, more specifically that if you hide the events that represent the low level signals (i.e. the wires), they are identical.

```
GEN_SPEC10 = PROC_PROCESS_CONTROL_ANNOTATED_OUTER_SPEC
```

Figure 145: Linking GenSpec 10 to GenSpec 5

H.2.15 GenSpec Assertions 15: Linking GenSpec 6 to GenSpec 9

These assertions (see Figure 146) demonstrate that properties of GenSpec 6 are similar to GenSpec 7. This proves the link between the clock cycle based higher annotation specifications and the software based higher annotation specification (non-clocked).

```
-- The annotation index value being used
annotation_id = 0

-- An instance of the annotation specification
GEN_SPEC6 = PROC_PROCESS_HIGHER_OUTER_SPEC(annotation_id)

-- Checking the annotation specification while ensuring that reset does
-- not occur (running in parallel with STOP), and hiding clock cycle annotations
EXPECTED_SIMPLIFIED_ANNOTATION_SPEC =

   (GEN_SPEC6
    [\ { annotation.RESET | ] ]
    STOP
   ) \ { annotation.x | x<-{ IDLE, NOTFINISHED, NOTRESET } ] ]

-- The new simplified specification to check against, GenSpec7
SIMPLIFIED_ANNOTATION_SPEC = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(annotation_id)

-- Check that the two processes are equivalent
assert EXPECTED_SIMPLIFIED_ANNOTATION_SPEC [P= SIMPLIFIED_ANNOTATION_SPEC
assert SIMPLIFIED_ANNOTATION_SPEC [T= EXPECTED_SIMPLIFIED_ANNOTATION_SPEC
```

Figure 146: Example of how to link Annotation Spec to Simplified Spec

H.3 Conclusion & Evaluation

The combination of the assertions covered in section H.2 links various properties of the various models covered in section H.1. This builds up confidence with the models specified so that implemented components can both utilise them in their own proofs and be checked against them. It provides processes so that a low hardware level model of the components can be refinement checked against, a higher clock cycle based conceptual
specification to refinement check against, and a method to link the two types of models together to show consistency between them.

The modifications that have been applied, thus causing the work to differ from that covered in Appendix D, causes the higher level models to be modelled in a more serial manner. Where as previously the models were annotated such that it was more directly mimicking the hardware, components that triggered other internal components when started appeared to start in parallel. The changes have resulted in the annotations mimicking more closely how the initial software appears, thus the annotation model now appears such that the component is triggered just before the internal component is triggered (but within the same clock cycle). This change more closely resembles a model of the software language, where the internal component can only be triggered after the command it is contained within has been triggered (e.g. the test for an ‘IF’ block is only performed after the ‘IF’ block has been reached).
Appendix I  Alternative Implemented Model Example

This section will cover and explain the CSP models required for an implemented component for the future work described in Appendix H, along with the assertions that need to be checked to link the models to each other, thus demonstrating that an implemented component performs as required and within the behaviours dictated by its generic super-type component. This work is the same as that covered in Appendix E, with the main exception of the parts covered in sections I.1.4, some other minor modifications have been added to the models through the introduction of the 'NOTRESET' annotation. This was required so that when an annotated model of a component that has its internal signal events hidden (i.e. when the annotated model of the segment of logic circuit is compared against ImpSpec 4), the model performs correctly when it has been started, has not finished, and has not been reset.

I.1 Models & Specifications

The 'internalChoice' event that may appear within the code examples has been utilised instead of internal choice (i.e. '\( \sim I \)' ) to enable 'chase' compression to applied if desired. The 'internalChoice' event must be hidden for the specifications to be valid, but if 'chase' compression has been chosen, the event should only be hidden after 'chase' has been applied, otherwise the specification becomes invalid.

I.1.1 ImpSpec 1: Valid Low Level Behaviour

This CSP model (see Figure 147) which is similar to the model defined in section D.1.1, specifies all the valid and allowable low level behaviour that this implemented component may perform at its outer boundary. It may utilises internal choice to determine the possible output behaviour it can perform, although it is not a requirement (e.g. boolean true, boolean false, SKIP, STOP, all have well defined fixed behaviours that do not rely on other internal components). It is useful to note that some implemented components
may have the allowable interface boundary behaviour that is identical to that of its generic
super type component (e.g. boolean comparisons, PAR), where as other components will
have an interface boundary behaviour that is a refinement of its super type component
(e.g. boolean true, boolean false, SEQ).

\[
\text{PROC\_PROCESS\_IF\_DESIRED\_SPEC(clock, reset, start, finish) =}
\]
\[
\text{let}
\]
\[
A =
\text{start?x -> reset?y -> C(x, y)}
\]
\[
B =
\text{start?0 -> reset?y -> D(y)}
\]
\[
C(x, y) =
\text{clock?1 -> finish!0 ->}
\]
\[
D(y) =
\text{y == 1 & clock?1 -> finish!0 -> A}
\]
\[
\text{within finish!0 -> A}
\]

Figure 147: Low Level 'IF' Component Desired Specification

1.1.2 ImpSpec 2: Low Level Behaviour with Explicit Deadlocking

This CSP model (see Figure 148) is similar to the model covered in section D.1.2, the
CSP model is the model covered in section E.1.1 but altered so that it will accept
incorrectly driven input signals followed by explicitly defined deadlocking (i.e. STOP).
Appendix G: Multi-Type Component Implemented Model Example

PROC_PROCESS_IF_GENERIC_SPEC(clock, reset, start, finish) =
let
A =
  start?x -> reset?y -> C(x, y)
  reset?y -> start?x -> C(x, y)
B =
  start?1 -> STOP
  start?0 -> reset?y -> D(y)
  reset?y ->
    ( start?0 -> D(y)
      []
      start?1 -> STOP
    )
C(x, y) =
  clock?1 -> finish!0 ->
    ( y == 1 & A
      []
      y == 0 &
      ( x == 0 & A
        []
        x == 1 & B
      )
    )
D(y) =
  y == 1 & clock?1 -> finish!0 -> A
  []
  y == 0 & clock?1 ->
    ( internalChoice -> finish!1 -> A
      [] -- | |
      internalChoice -> finish!0 -> B
    )
within finish!0 -> A

Figure 148: Low Level 'IF' Component Generic Specification with Explicit Deadlocking

I.1.3 ImpSpec 3: Model of Implemented Logic

This CSP model (see Figure 149), is a model of the segment of logic that this component represents and is achieved through modelling the individual logic components and running them in parallel.

Figure 149: CSP Model of the Logic Circuit Segment of the Implemented 'IF' Component

```
-- Components used in the segment of logic being verified
alpha_PROC2 = { chan1, chan8, chan0, chan7 }
PROC2 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan7, chan8, 3)
alpha_PROC7 = { chan9, chan4 }
PROC7 = PROC_NOT(chan9, chan4)
alpha_PROC1 = { chan1, chan6, chan0, chan5 }
PROC1 = PROC_PROCESS_ANNOTATED_SPEC(chan0, chan1, chan5, chan6, 2)
alpha_PROC8 = { chan6, chan8, chan10 }
PROC8 = PROC_OR(chan10, chan6, chan8)
alpha_PROC9 = { chan9, chan3, chan7 }
PROC9 = PROC_AND(chan7, chan9, chan3)
```
Appendix G: Multi-Type Component Implemented Model Example

\[
\begin{align*}
\alpha_{\text{PROC10}} &= \{ \text{chan4, chan3, chan5} \} \\
\text{PROC10} &= \text{PROC\_AND(chan5, <chan4, chan3>)}
\end{align*}
\]

\[
\begin{align*}
\alpha_{\text{PROC0}} &= \{ \text{chan4, chan1, chan3, chan0, chan2} \} \\
\text{PROC0} &= \text{PROC\_BOOLEAN\_ANNOTATED\_SPEC(chan0, chan1, chan2, chan3, chan4, 1)}
\end{align*}
\]

\[\text{SYSTEM\_INTERFACE} = \{ \text{chan1, chan0, chan10, chan2} \}\]

\[\text{SYSTEM\_ALPHA} = \{ \text{chan3, chan0, chan5, chan2, chan4, chan8, chan9, chan10, chan1, chan7} \}\]

\[\text{SYSTEM\_LIST} = < (\text{PROC2, } \alpha_{\text{PROC2}}), (\text{PROC7, } \alpha_{\text{PROC7}}), (\text{PROC1, } \alpha_{\text{PROC1}}), (\text{PROC8, } \alpha_{\text{PROC8}}), (\text{PROC9, } \alpha_{\text{PROC9}}), (\text{PROC10, } \alpha_{\text{PROC10}}), (\text{PROC0, } \alpha_{\text{PROC0}}) >\]

\[\text{SYSTEM} = \text{REPL(SYSTEM\_LIST)} \setminus \text{diff(SYSTEM\_ALPHA, SYSTEM\_INTERFACE)}\]

\[\text{REPL(p)} = \]
let
\[
\begin{align*}
\text{INNER1}(p1, a1, p2, a2) &= p1 \setminus \text{inter}(a1, a2) \setminus p2 \\
\text{INNER2}(<(p1, a1)>, <(p2, a2)>)^p3 &= \text{null}(p3) \& \text{INNER1}(p1, a1, p2, a2) \\
\text{INNER3}(<(p1, _)>^p) &= p1
\end{align*}
\]
within (null(p) \& STOP [])
length(p) == 1 \& INNER3(p) []
length(p) > 1 \& INNER2(p) 
\]

A graphical depiction of the segment of logic circuit that this model represents can be seen in Figure 150.
I.1.4 ImpSpec 4: Annotation Only Specification

This CSP model (see Figure 151), is the expected behaviour of the ‘IF’ component from an annotation only perspective, with annotation models of the internal component (i.e. the boolean condition, the ‘then’ process and the ‘else’ process) having to be supplied. If the internal annotation components supplied are the corresponding generic specifications, the CSP model will demonstrate all the possible behaviours of the ‘IF’ component, describing both how driving the component drives the internal components and the internal components behaviour effects the outputs of this component. The reason why this model was designed to take in processes representing the internal components is so that if the supplied internal component specifications are a refinement of the corresponding generic specification, they will limit the behaviour dictated in the ‘IF’ component to that which describes what should conceptually occur in the hardware.

Figure 151: IF Component Annotation Only Specification

```
channel chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC : 0..2
channel chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC : {0, 1}
PROC_PROC_PROCESS_IF_HIGHER_INNER_SPEC(id, bool, bid, thenproc, tid, elseproc, eid) = 
let
  A =
  ( ( ||| x:{bool, thenproc, elseproc} @ x )
  || { annotation.z.x,
  annotation.y.bid,
  annotation.w.v
  | vc={tid, eid},
  xc={bid, tid, eid},
  yc={BOOLEANREAD,
  BOOLEANREADNOTREADY,
  BOOLEANREADALLOWED,
  BOOLEANTRUE,
  BOOLEANFALSE},
  zc={RESET, NOTRESET, IDLE},
  wc={START, FINISH, NOTFINISHED}
  }
  |
  )

B =
-- combination of boolean and process
```
Appendix G: Multi-Type Component Implemented Model Example

```plaintext
( annotation.RESET.id ->
  ( ||| x:{bid, tid, eif} & annotation.RESET.x -> SKIP )
) || annotation.IDLE.id ->
  ( ||| x:{bid, tid, eif} & annotation.IDLE.x -> SKIP )
) || annotation.START.id ->
  ( C
    || chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
    annotation.RESET.id,
    annotation.NOTRESET.id |
  ) |
( annotation.BOOLEANREAD.bid ->
  ( annotation.BOOLEANDONTREAD.bid ->
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> D
  ) |
  annotation.BOOLEANREADALLOWED.bid ->
  ( annotation.BOOLEANFALSE.bid ->
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 -> E
  ) |
  annotation.BOOLEANTRUE.bid ->
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> E
  )
)
)
)
;
B
-- Then & Else processes
C =
  let
    CA =
      ( ( ||| x:{tid, eid} & annotation.IDLE.x -> SKIP )
        ||| ( annotation.NOTFINISHED.id -> SKIP )
      ) |
      chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> CB
    ) |
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> CC(tid, eid)
    ) |
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 -> CC(eid, tid)
  CB =
    annotation.RESET.id ->
      ( ||| x:{tid, eid} & annotation.RESET.x -> SKIP )
    ) |
    annotation.NOTRESET.id -> CA
CC(x, y) =
  let
    CCA =
      ( ( CCB
        || chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
        annotation.RESET.id,
        annotation.NOTRESET.id |
      )
    ) |
    CCC
  ) |
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
    annotation.RESET.id,
    annotation.NOTRESET.id |
  ) |
  CCB
```
Appendix G: Multi-Type Component Implemented Model Example

CCB =
  annotation.RESET.id -> annotation.RESET.y -> SKIP
  []
  annotation.NOTRESET.id -> annotation.IDLE.y ->
    { chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> CCB
    []
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> SKIP
  }
CCC =
  annotation.RESET.id -> annotation.RESET.x -> SKIP
  []
  annotation.NOTRESET.id -> annotation.START.y ->
    { annotation.FINISH.y ->
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> SKIP
    []
    annotation.NOTFINISHED.y ->
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> CCD
  }
CCD =
  annotation.RESET.id -> annotation.RESET.x -> SKIP
  []
  annotation.NOTRESET.id ->
    { annotation.FINISH.id -> SKIP
    []
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> CCD
  }

within CCA
  within CA
    -- Segment of Boolean Process

D =
  annotation.RESET.id -> annotation.RESET.bid -> SKIP
  []
  annotation.NOTRESET.id ->
    { annotation.BOOLEANDONTREAD_bid ->
    chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> D
    []
    annotation.BOOLEANREADALLOWED.bid ->
      annotation.BOOLEANFALSE.bid ->
        chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.2 -> E
    []
    annotation.BOOLEANTRUE.bid ->
      chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> E
    }
E =
  annotation.RESET.id -> annotation.RESET.bid -> SKIP
  []
  annotation.NOTRESET.id -> annotation.IDLE.bid ->
    { chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.1 -> SKIP
    []
    chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC.0 -> E
  }
within ( A \ [ ]
  { [ ]
  chan_PROC_PROCESS_IF_HIGHER_INNER_SPEC,
  chan_fin_PROC_PROCESS_IF_HIGHER_INNER_SPEC
} )

289
I.1.5 ImpSpec 5: Software Specification Model

This model (see Figure 152) provides a software level based model of the implemented component. The external choice with 'STOP' is to provide a clear indication of where the internal components behaviour is expected to create possible deadlocking within this model. The deadlocking at within this model is allowed to possibly occur under the conditions where when an internal component is started, it never completes. Should this condition arise, the 'IF' component will never finish, a simple example of this is if the 'then' component gets triggered and is a 'while(true)' loop, the 'while(true)' loop never finishes, and so the 'IF' component would never finish.

```
PROC_PROCESS_IP_HIGHER_SIMPLIFIED_SPEC(id,
   bool, bid,
   thenproc, tid,
   elseproc, eid) =

let
  A =
  ( annotation.START.id -> SKIP
    ||
    annotation.BOOLEANREAD.bid -> SKIP
  )
  ;
  ( STOP
    []
    annotation.BOOLEANREADALLOWED.bid ->
      ( annotation.BOOLEANTRUE.bid -> B(tid)
        []
        annotation.BOOLEANFALSE.bid -> B(eid)
      )
  )
  B(x) =
    annotation.START.x ->
      ( STOP
        []
        annotation.FINISH.x -> A
      )
  C =
    ( ||{ x:{bool, thenproc, elseproc} • x}
      ||{ annotation.BOOLEANREAD.bid,
        annotation.BOOLEANREADALLOWED.bid,
        annotation.BOOLEANTRUE.bid,
        annotation.BOOLEANFALSE.bid,
        annotation.START.y,
        annotation.FINISH.y,
        | y <-{tid, eid} }
    )
  )
  A
  within C
```

Figure 152: 'IF' Component Software Annotation Behavioural Specification
I.2 Assertions: Linking the Models Together

I.2.1 ImpSpec Assertion 1: Deadlock-Free

The assertion stated in Figure 153 checks that the model of the segment of logic circuit is deadlock-free. This check demonstrates three properties, the first is that there exists no loops consisting of only clocked or non-clocked logic components, secondly is that any internal components are guaranteed to be driven correctly so long as the outer component is driven correctly, and thirdly is that the component deal with all valid inputs of its super type component. The guarantee that internal components are driven correctly is possible because the models of the internal components are models that accept all possible inputs (both valid and incorrect), with the incorrect inputs being followed by explicitly defined deadlock i.e. 'STOP' (see section H.1.2). If the STOP's are not reached, then invalid inputs to internal components have not been created or propagated through. The control process used to drive the implemented component is the control process from its super type. As the control process (see section H.1.3) utilises internal choice to provide all the possible valid driving signals, if the deadlock-free assertion holds true then the implementation is guaranteed to be able to process all patterns of driving signals that it may receive, along with the guarantee that it will not produce incorrect output signals.

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
-- The alpha_PROC contains the low level channels used by the processes
alpha_PROC6 = {{ chan1, chan0, chan10, chan2 }}
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = {{ chan1, chan0, chan10, chan2 }}
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

-- Deadlock-free check the expected correct generic component
assert (IMP_SPEC3 [[[ SYSTEM_INTERFACE ]]] GEN_SPEC3) : (deadlock free [F])
```

Figure 153: 'IF' Component Deadlock-Free Assertion
1.2.2 ImpSpec Assertion 2: Super Type Control Only Limits the Behaviour

The assertion stated in Figure 154 demonstrates that the process that dictates and provides
the allowable correct driving input signals of the super type of this component, does only
limit the behaviour of this implemented component, thus ensuring that it does not
introduce any new behaviour.

```
-- channel declarations
channel internalchoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
-- The alpha PROC contains the low level channels used by the processes
alpha PROC6 = {{chan1, chan0, chan10, chan2}}
PROC6 = PROC_PROCESS_CONTROLL(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = {{chan1, chan0, chan10, chan2}}
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6

-- Check that the control specification only limits the behaviour of the segment
-- of logic, and does not introduce new behaviour
assert IMP_SPEC3 [T (IMP_SPEC3 [F SYSTEM_INTERFACE |] GEN_SPEC3)
```

Figure 154: Super Type Component Limits the Behaviour of the Implementation

1.2.3 ImpSpec Assertion 3: Expected 'IF' Component Behaviour is
Deadlock-Free

The assertion stated in Figure 155 demonstrates that the expected boundary behaviour
that the model of the implementation of the logic circuit segment will be checked against
is deadlock-free. This is to provide better confidence in this models correctness as it will
be used in future checks.

```
-- channel declarations
channel internalchoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
assert IMP_SPEC1 : [deadlock-free [F]]
```

Figure 155: Expected 'IF' Component Behaviour is Deadlock-Free
I.2.4 ImpSpec Assertion 4: Expected Boundary Behaviour Refines Super Type

The assertion stated in Figure 156 demonstrates that the expected correct boundary behaviour of the implemented component is a valid refinement of the expected boundary behaviour of its generic super type component.

```plaintext
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)
GEN_SPEC1 =
    ( PROC_PROCESS_DESIRED_GENERIC_SPEC(chan0, chan1, chan2, chan10)
      \ {internalChoice}
    )
assert GEN_SPEC1 [T= IMP_SPEC1
```

Figure 156: Expected 'IF' Component Behaviour is a Refinement of Super Type

I.2.5 ImpSpec Assertions 5: Correctly Driven Implementation Behaves as Expected

The assertions stated in Figure 157 demonstrate that the segment of logic circuit for this implemented component, if driven correctly, behaves as expected. Trace refinement ('T=') is used instead of failures divergent ('FD=') because the process used to drive the model of the segment of logic utilises internal choice to provide all possible allowed driving signals, where as the model of the expected behaviour uses external choice for the input signals because the input signals are provided by an outside source.
Appendix G: Multi-Type Component Implemented Model Example

**Figure 157: Correctly Driven 'IF' Component Behaves as Expected**

I.2.6 ImpSpec Assertion 6: Annotation Outer Level Does Not Introduce Deadlock

The assertion stated in Figure 158 demonstrates that using the annotated control to annotate the outer level of the implemented component, with this process being specified by its super type (i.e. GenSpec 5, see Section D.1.5), does not introduce any deadlocks.
1.2.7 ImpSpec Assertion 7: Expected High Level Behaviour is Deadlock-Free

The assertion stated in Figure 159 demonstrates that the high level model describing the expected behaviour of the implemented component is deadlock-free. This helps to build confidence in the model for when using it in future checks.

```plaintext
-- Create an instance of the models to check
IMP_SPEC4 = PROC_PROCESS_IF_HIGHER.Inner_SPEC(0,
           HIGHER_SPEC0_1, 1,
           HIGHER_SPEC1_2, 2,
           HIGHER_SPEC2_3, 3)

-- Higher Process Instances
HIGHER_SPEC1_2 = PROC_PROCESS.HIGHER. OUTER_SPEC(2)
HIGHER_SPEC2_3 = PROC_PROCESS.HIGHER. OUTER_SPEC(3)
HIGHER_SPEC0_1 = PROC.BOOLEAN.HIGHER. OUTER_SPEC(1)
assert IMP_SPEC4 ; [deadlock free[P]]
```

Figure 159: 'IF' Component High Level Behaviour is Deadlock-Free

1.2.8 ImpSpec Assertions 8: Component Behaves Similarly to Expected Higher Spec

The assertions stated in Figure 160 demonstrate that the annotations obtained from the implemented segment of logic circuit, performs in a similar manner to that of the expected higher behavioural specification. The test can not be failure divergence checked both ways (one has to be a trace refinement), this is due to the way annotations are added to the outer layer. As the outer level input annotations occur after the corresponding low level input signal events (i.e. the events that represent the wires), hiding these low level signal events causes the high level model extracted from the implemented segment of logic circuit to appear to have internal choice determining the high level conceptual input states. The internal choice for the inputs does not really exist, but appears because the events that do determine what occurs though external choice have been hidden (i.e. the low level signals). Through altering the process of annotating the outer level of a component (see Appendix H), it is possible to simplify the extracted model so that it directly equivalent to the expected higher behaviour.
Appendix G: Multi-Type Component Implemented Model Example

```
-- channel declarations
channel internalChoice
channel chan0 : {1}
channel chan1 : {0, 1}
channel chan2 : {0, 1}
channel chan10 : {0, 1}

-- Create an instance of the models to check
IMP_SPEC1 = PROC_PROCESS_IF_DESIRED_SPEC(chan0, chan1, chan2, chan10)

-- Create an instance of the models to check
The alpha PROC contains the low level channels used by the processes
alpha_PROC6 = {{ chan1, chan0, chan10, chan2 }}
PROC6 = PROC_PROCESS_CONTROL(chan0, chan1, chan2, chan10)

alpha_PROC5 = {{ chan1, chan0, chan10, chan2 }}
PROC5 = PROC_PROCESS_ANNOTATE OUTER(chan0, chan1, chan2, chan10)

-- An implementation of the logic to check
SYSTEM_INTERFACE = {{ chan1, chan0, chan10, chan2 }}
IMP_SPEC3 = SYSTEM
GEN_SPEC3 = PROC6
GEN_SPEC5 = PROC5

assert (( (IMP_SPEC3 || SYSTEM_INTERFACE || GEN_SPEC3)
           || SYSTEM_INTERFACE || GEN_SPEC5 ) \ SYSTEM_INTERFACE )
assert Imp_Model_3 [T= (( (IMP_SPEC3 || SYSTEM_INTERFACE || GEN_SPEC3)
           || SYSTEM_INTERFACE || GEN_SPEC5 ) \ SYSTEM_INTERFACE )
```

Figure 160: 'IF' Component Behaves Similarly to Expected Higher Behaviour

1.2.9 ImpSpec Assertion 9: Software Specification is a Refinement of Super Type

The assertion stated in Figure 161 demonstrates that the expected higher software specification for the implemented component with its internal events hidden is a refinement of its super types' software specification model. The example show happens to be equivalent to its super type software specification model, but this is not a requirement and is why it is not being tested for.
Appendix G: Multi-Type Component Implemented Model Example

```lisp
-- Generic Boolean Higher Software Specification
PROC_BOOLEAN_HIGHER_SIMPLIFIED_SPEC(id) =

let

    A =
        annotation.BOOLEAN_READ_ALLOWED.id ->
        ( STOP
          |-
        annotation.BOOLEAN_READ_ALLOWED.id ->
        ( annotation.BOOLEAN_TRUE.id -> A
          |
          annotation.BOOLEAN_FALSE.id -> A
        )
    )

Within A

-- Internal Components
BoolTest = PROC_BOOLEAN_HIGHER_SIMPLIFIED_SPEC(1)
ThenProc = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(2)
ElseProc = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(3)

-- Components to Test
IMP_SPEC5 = PROC_PROCESS_IF_HIGHER_SIMPLIFIED_SPEC(0,
                        BoolTest, 1,
                        ThenProc, 2,
                        ElseProc, 3)

GEN_SPEC7 = PROC_PROCESS_HIGHER_SIMPLIFIED_SPEC(0)
IMP_SPEC5_HIDDEN_INTERNALS =
    ( IMP_SPEC5
      \ diff( [ | annotations |],
             { annotations.x.0 | x <- { START, FINISH } } )
    )

-- Check that ImpoSpec5 with internal events hidden is a refinement of its
-- super type
assert GEN_SPEC7 \= IMP_SPEC5_HIDDEN_INTERNALS

Figure 161: Higher Software Specification is a Refinement of the Super Type
```
Appendix J  Pattern Matched Selective Renaming Alternative

This section of work covers a methodology to circumvent the lack of ability to perform event renaming constrained by pattern matching. The ability to perform pattern matched event renaming would simplify the process of annotating and extracting the conceptual meanings for occurrences of events within a low level model, without the need to refactor and modify the models.

Currently the technique has not been adapted to deal with describing the conceptual meanings that are dependent on the combination of multiple parallel events. The “IF THEN ELSE” component has a combination of parallel events sent along its start and reset channels determining if the process starts, has been reset, or an error has occurred due to incorrect driving. It is due to this feature why the example that is demonstrated is a simplification of the “IF THEN ELSE” component with the reset functionality removed (see Figure 162 and Figure 163). A further requirement for this technique to work properly is that internal OCCAM components can not directly utilise the same signals of that which are used at the boundaries for the components they lie within and results in the an addition of a single input OR gate added into the logic circuit (see Figure 162 for the yellow OR gate). This is required so that when the refactored model of the logic circuit has selective low level events replaced with annotation events, the CSP can synchronise correctly on the events. If the internal OCCAM components utilised the same events as the boundary components they lie within, instances of the low level events would be renamed to different annotation events in different models, thus greatly complicating the process of integrating them together. The addition of extra single input OR gates has no adverse effect of the logic circuits being created for two reasons, the first is that from a clock cycle perspective the extra OR gate does not introduce clock cycle delays. The second reason why the extra OR gate does not impact on the design is that it is easy to optimised away after the whole logic circuit has been generated, thus it has no impact on the speed that the clock will be able to run at.
Figure 162: Graphical representation of simplified "IF THEN ELSE" component
Appendix J: Pattern Matched Selective Renaming Alternative

-- Components used in the segment of logic being verified
alpha_PROC2 = { chan8, chan0, chan7 }
PROC2 = PROC_PROCESS_GENERIC_SPEC_NORESET(chan0, chan7, chan8)
alpha_PROC7 = { chan9, chan4 }
PROC7 = PROC_NOT(chan9, chan4)
alpha_PROC1 = { chan6, chan0, chan5 }
PROC1 = PROC_PROCESS_GENERIC_SPEC_NORESET(chan0, chan5, chan6)
alpha_PROC8 = { chan6, chan8, chan10 }
PROC8 = PROC_OR(chan10, chan6, chan8)
alpha_PROC9 = { chan9, chan3, chan7 }
PROC9 = PROC_AND(chan7, chan9, chan3)
alpha_PROC10 = { chan4, chan3, chan5 }
PROC10 = PROC_AND(chan5, chan4, chan3)
alpha_PROC11 = { chan11, chan2 }
PROC11 = PROC_OR(chan11, chan2)
alpha_PROC0 = { chan4, chan3, chan0, chan11 }
PROC0 = PROC_BOOLEAN_GENERIC_SPEC_NORESET(chan0, chan11, chan3, chan4)

-- The outer level signals for the component being tested
SYSTEM_INTERFACE = { chan0, chan10, chan2 }
-- The alphabet of signals used in the model of the logic
SYSTEM_ALPHA = { chan3, chan0, chan6, chan2, chan5, chan4, chan8, chan9, chan10, chan11, chan7 }

-- System Declaration of the internal logic components and the signals they use
SYSTEM_LIST = ( PROC2, alpha_PROC2 , PROC7, alpha_PROC7 , PROC1, alpha_PROC1 , PROC8, alpha_PROC8 , PROC9, alpha_PROC9 , PROC10, alpha_PROC10 , PROC11, alpha_PROC11 , PROC0, alpha_PROC0 )

-- The logic model of the implemented component. 'ImpSpec3'
SYSTEM_WITH_INTERNALS = REPL2(SYSTEM_LIST) \ {internalChoice }

-- Used to run the logic components in parallel
REPL2(p) =
let
INNER1(p1, a1, p2, a2) = p1 [ inter(a1, a2) | p2
INNER2( <p1,a1> ^p2,a2 ) =
null(p3) & INNER1(p1,a1,p2,a2)
[]
not null(p3) & INNER2( union(a1,a2), union(a1,a2) )
INNER3( <p1, _ > ) = p1
within ( null(p) & STOP
[]
length(p) == 1 & INNER3(p)
[]
length(p) > 1 & INNER2(p) )

Figure 163: CSP Model of "IF THEN ELSE" Component
Appendix J: Pattern Matched Selective Renaming Alternative

J.1 Methodology

The alternative methodology to pattern matched event renaming requires the creation and linkage (through assertions) of various models. The initial stage involves the specification defining the expected low level behaviour of the model of the segment of circuit (see Figure 164). This specification can be perceived as a refactored model containing the same behaviour as that of the segment of logic circuit constrained with being correctly driven (see Figure 165).

Figure 164: Refactored correctly driven circuit

```
channel chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET : [0..5]
channel chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET : [0..3]
PROC_PROCESS_IF_REFACTORED_SPEC_NORESET(
  clock, start, finish,
  boolproc, boolstart, boolfin, boolstate,
  thenproc, thenstart, thenfin,
  elseproc, elsestart, elsefin,
  negation
) =
  let
    -- Perform checks on the input parameters
    A =
      inter({ start, finish}
        |{ boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin
           ) == {} & B
      [])
    B =
      inter({ start, finish}
        |{ boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin
           ) == {} & STOP
     -- Perform the model
    -- Run the internal components in parallel with the remainder of the IF -- component
    E =
      (( x: {boolproc, thenproc, elseproc} @
      ( x \ {internalChoice})) )
      ( [ ]
        |{ boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin
           )
      )
    C =
      let
        CA =
          ( ( thenfin?0 -> SKIP )
            ( elsefin?0 -> SKIP )
          )
          ( finish?0 ->
            ( start?0 ->
              boolstart?0 ->
          channel-chan_PROC_PROCESS_IP_REFACTORED_SPEC_NORESET.0 ->
```

301
Appendix J: Pattern Matched Selective Renaming Alternative

CA
[
start?1 ->
 boolstart!1 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 ->
 CB
 ]
}

CB =
( ( thenfin?0 -> SKIP )
| |
( elsefin?0 -> SKIP )
)

( finish!0 -> start?0 -> boolstart!0 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.2 -> CB
 []
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.3 -> CC
 []
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.4 -> CD
 )

CC =
chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 ->
( thenfin?0 ->
 chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 ->
 finish!0 ->
 start?0 ->
 boolstart!0 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.5 ->
 CC
 [])
thenfin?1 ->
 chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 ->
 finish!1 ->
 ( start?0 ->
 boolstart!0 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 ->
 CA
 [])
start?1 ->
 boolstart!1 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 ->
 CB
)

CD =
chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.2 ->
( elsefin?0 ->
 chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.3 ->
 finish!0 ->
 start?0 ->
 boolstart!0 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.5 ->
 CD
 [])
elsefin?1 ->
 chan_link_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.3 ->
 finish!1 ->
 ( start?0 ->
 boolstart!0 ->
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 ->
 CA
 [])
start?1 ->
 boolstart!1 ->

302
Appendix J: Pattern Matched Selective Renaming Alternative

```plaintext
chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> CB
)

within CA
--
D =
let
   DA =
   boolfin?0 ->
   boolstate?0 ->
   ( negation.1 ->
      ( ( thenstart!0 -> SKIP )
         |||
         ( elsestart!0 -> SKIP )
      )
      ;
      ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 -> DA
        []
        chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> DB
      )
   )

[]
( ( thenstart!0 -> SKIP )
   |||
   ( negation.1 -> elsestart!0 -> SKIP )
 )
;
( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 -> DA
  []
  chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> DB
 )
)

DB =
   boolfin?0 ->
   boolstate?0 ->
   ( negation.1 ->
      [ [ ( thenstart!0 -> SKIP )
         |||
         ( elsestart!0 -> SKIP )
      ]
      ;
      ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.2 -> DB )
   )

[]
( ( thenstart!0 -> SKIP )
   |||
   ( negation.1 -> elsestart!0 -> SKIP )
 )
;
( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.2 -> DB )
)
)

[]
boolfin?1 ->
( boolstate?0 ->
   ( negation.1 ->
      [ [ ( thenstart!0 -> SKIP )
         |||
         ( elsestart!1 -> SKIP )
      ]
      ;
      ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.4 -> DD )
   )

[]
( ( thenstart!0 -> SKIP )
   |||

303
```
Appendix J: Pattern Matched Selective Renaming Alternative

( negation.1 -> elsestart!1 -> SKIP )

; ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.4 -> DD )

)

[]
boolstate!1 ->
( negation.0 ->
  ( ( ( thenstart!1 -> SKIP )
     |||
     ( elsestart!0 -> SKIP )
   )

   ;
   ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.3 -> DC )
  )

[]
( ( thenstart!1 -> SKIP )
  |||
  ( negation.0 -> elsestart!0 -> SKIP )
)

;
( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.3 -> DC )
)

)

DC =
boolfin?0 ->
boolstate?0 ->
( negation.1 ->
  ( ( ( thenstart!0 -> SKIP )
     |||
     ( elsestart!0 -> SKIP )
   )

   ;
   ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.5 -> DC
     []
   chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 -> DA
     []
   chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> DB
   )

)[
( ( thenstart!0 -> SKIP )
  |||
  ( negation.1 -> elsestart!0 -> SKIP )
)

;
( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.5 -> DC
 []
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 -> DA
 []
 chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> DB
 )

)

)

DD =
boolfin?0 ->
boolstate?0 ->
( negation.1 ->
  ( ( ( thenstart!0 -> SKIP )
     |||
     ( elsestart!0 -> SKIP )
   )

   ;
   ( chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.5 -> DC
     []
   chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.0 -> DA
     []
   chan_PROC_PROCESS_IF_REFACTORED_SPEC_NORESET.1 -> DB
   )

)

)
Appendix J: Pattern Matched Selective Renaming Alternative

Figure 165: Assertions demonstrating low level specification equivalence

```plaintext
SYSTEM_INTERFACE = [ chan0, chan10, chan2 ]
SYSTEM_CONTROL = PROC_PROCESS_CONTROL_NORESET(chan0, chan2, chan10)
SYSTEM_WITH_CONTROL =
  (SYSTEM_WITH_INTERNALS [ SYSTEM_INTERFACE ] SYSTEM_CONTROL)
LOW_LEVEL_SPEC = PROC_PROCESS_IF_REFACTORED_SPEC_NORESET(
  chan0, chan2, chan10,
  PROCO, chan11, chan3, chan4,
  PROC1, chan5, chan6,
  PROC2, chan7, chan8,
  chan9
)
assert LOW_LEVEL_SPEC [FD= SYSTEM_WITH_CONTROL]
assert SYSTEM_WITH_CONTROL [FD= LOW_LEVEL_SPEC]
```
Appendix J: Pattern Matched Selective Renaming Alternative

The methodology involves adapting the specifications through replacing selected events with annotation events, thus describing the conceptual meaning for specific instances the events have. See Figure 168 for the modified "IF" component specification, Figure 166 for the modified generic control flow process specification and Figure 167 for the modified generic boolean process specification. Renaming in its current state can not be used to achieve these models, as all occurrences of a low level event may not conceptually mean the same thing and thus global replacing can not be used on a process.

```
-- This process was modified from PROC_PROCESS_DESIRED_GENERIC_SPEC_NORESET
-- See Figure 175 on page 6
PROC_PROCESS_GENERIC_HYBRID_SPEC_NORESET(clock, start, finish, id) =
   let
      A =
          annotation.START.id -> clock?1 ->
          ( internalChoice -> annotation.FINISH.id -> A
            [] -- []
            internalChoice -> annotation.NOTFINISHED.id -> B
          )
      []
      annotation.IDLE.id -> clock?1 -> finish!0 -> A
      B =
          start?1 -> STOP
          []
          start?0 -> clock?1 ->
          ( internalChoice -> annotation.FINISH.id -> A
            [] -- []
            internalChoice -> annotation.NOTFINISHED.id -> B
          )
      within finish!0 -> A

Figure 166: Hybrid Generic Control Flow Process
```

```
-- This process was modified from PROC_BOOLEAN_GENERIC_SPEC_NORESET
-- See Figure 178 on page 8
PROC_BOOLEAN_GENERIC_HYBRID_SPEC_NORESET(clock, bool_read, bool_readallowed, state, id) =
   let
      A =
          bool_readallowed!0 -> state!0 -> B
      B =
          annotation.IDLE.id -> clock?1 -> A
      []
          annotation.BOOLEANREAD.id -> clock?1 -> D
      D =
          internalChoice -> annotation.BOOLEANREADNOTREAD.id -> state!0 -> E
      [] -- []
          internalChoice -> annotation.BOOLEANREADALLOWED.id ->
          ( internalChoice -> annotation.BOOLEANFALSE.id -> B
            [] -- []
            internalChoice -> annotation.BOOLEANTRUE.id -> B
          )
      E =
          bool_read?1 -> STOP
      []
          bool_read?0 -> clock?1 -> D
      within A

Figure 167: Hybrid Generic Boolean Process
```
Appendix J: Pattern Matched Selective Renaming Alternative

Figure 168: Low Level 'IF' Component Specification with Embedded Full Annotations

```
channel chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET : [0..5]
channel chan_link_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET : [0..3]
PROCPROCESS_IFHYBRIDFULLSPEC_NORESET(
    clock, start, finish, id,
    boolproc, boolstart, boolfin, boolstate, bid,
    thenproc, thenstart, thenfin, tid,
    elseproc, elsestart, elsefin, eid,
    negation
) =
    let
        -- Perform checks on the input parameters
        A =
            inter( [ start, finish]
            ,
                [ boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin
                ]
            ) == {} & B
            inter( [ start, finish]
            ,
                [ boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin
                ]
            ) != {} & STOP
        -- Perform the model
        -- Run the internal components in parallel with the remainder of the IF
        -- component
        B =
            ( [ ] [ ] [ ] x: {boolproc, thenproc, elseproc} @
                x \ {internalChoice} )
            [ ] [ ] [ ]
            ( clock, boolstart, boolfin, boolstate, thenstart, thenfin, elsestart, elsefin,
                annotation.START.x,
                annotation.FINISH.x,
                annotation.NOTFINISHED.x,
                annotation.IDLE.y,
                annotation.BOOLEANREAD.bid,
                annotation.BOOLEANREADTRUE.bid,
                annotation.BOOLEANREADALLOWED.bid,
                annotation.BOOLEANFALSE.bid,
                annotation.BOOLEANDONTREAD.bid
                )
            x<-{tid, eid}, y<-{bid, tid, eid} )
        )
        E =
    )
    C =
    let
        CA =
            ( [ ] [ ] [ ] [ ] [ ])
            ( thenfin?0 -> SKIP )
            ( elsefin?0 -> SKIP )
            )
            ( finish?0 ->
                [ annotation.IDLE.id ->
                    annotation.IDLE.bid ->
                    chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.0 ->
                    CA
                []
                annotation.START.id ->
                annotation.BOOLEANREAD.bid ->
                chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.1 ->
                CB
            )
```

307
Appendix J: Pattern Matched Selective Renaming Alternative

```plaintext
CB =
  ( ( thenfin? 0 -> SKIP )
    [ ]
    ( elsefin? 0 -> SKIP )
  )

!(( thenfin? O -> SKIP
  III ( elsefin? O -> SKIP

annotation. NOTFINISHED.id -> start? 0 -> boolstart!0 ->
  ( chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.2 -> CB
  [ ]
  chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.3 -> CC
  [ ]
  chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.4 -> CD

)

CC =
  chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.0 ->
  ( annotation.NOTFINISHED.tid ->
    chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.1 ->
    annotation.NOTFINISHED.id ->
    start? 0 ->
    annotation.IDLE.bid ->
    chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.5 ->
    CC
  )

[]

annotation.FINISH.tid ->
  chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.1 ->
  annotation.FINISH.id ->
  ( annotation.IDLE.id ->
    annotation.IDLE.bid ->
    chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.0 ->
    CA
  )

[]

annotation.START.id ->
  annotation.BOOLEANREAD.bid ->
  chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.1 ->
  CB

)

CD =
  chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.2 ->
  ( annotation.NOTFINISHED.oid ->
    chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.3 ->
    annotation.NOTFINISHED.id ->
    start? 0 ->
    annotation.IDLE.bid ->
    chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.5 ->
    CD
  )

[]

annotation.FINISH.oid ->
  chan_link_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.3 ->
  annotation.FINISH.id ->
  ( annotation.IDLE.id ->
    annotation.IDLE.bid ->
    chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.0 ->
    CA
  )

[]

annotation.START.id ->
  annotation.BOOLEANREAD.bid ->
  chan_PROC_PROCESS_IPHYBRID_FULL_SPEC_NORESET.1 ->
  CB

)

within CA
```

D =
```
let
  DA =
  boolfin?0 ->
  boolstate?0 ->
  ( negation.1 ->
    ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
    \{ annotation.IDLE.eid -> SKIP \}
    )
  )
  ;
  ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.0 -> DA
    []
    chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.1 -> DB
    )
  )
  [][]
  ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
    \{ negation.1 -> annotation.IDLE.eid -> SKIP \}
    )
  ;
  ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.0 -> DA
    []
    chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.1 -> DB
    )
  )
  ]
  DB =
  annotation.BOOLEANREAD.bid ->
  boolstate?0 ->
  ( negation.1 ->
    ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
    \{ annotation.IDLE.eid -> SKIP \}
    )
  )
  [][]
  ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
    \{ negation.1 -> annotation.IDLE.eid -> SKIP \}
    )
  ;
  ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.2 -> DB
    )
  )
  []
  annotation.BOOLEANREAD.bid ->
  ( annotation.BOOLEANFALSE.bid ->
    ( negation.1 ->
      ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
      \{ annotation.START.eid -> SKIP \}
      )
    )
  )
  [][]
  ( \{ \{ annotation.IDLE.tid -> SKIP \} \}
    \{ negation.1 -> annotation.START.eid -> SKIP \}
    )
  ;
  ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.4 -> DD
    )
  )
  []
Appendix J: Pattern Matched Selective Renaming Alternative

```
annotation.BOOLEANTRUE.bid ->
  ( negation.0 ->
    ( ( ( annotation.START.tid -> SKIP )
      |||
      ( annotation.IDLE.eid -> SKIP )
    )
    ;
    ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.3 -> DC )
  )
[]
( ( annotation.START.tid -> SKIP )
  |||
  ( negation.0 -> annotation.IDLE.eid -> SKIP )
)
( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.3 -> DC )

DD =
  boolfin?0 ->
  boolstate?0 ->
  ( negation.1 ->
    ( ( ( thenstart!0 -> SKIP )
      |||
      ( annotation.IDLE.eid -> SKIP )
    )
    ;
    ( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.5 -> DC
      []
      chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.0 -> DA
      []
      chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.1 -> DB
    )
  )
[]
( ( thenstart!0 -> SKIP )
  |||
  ( negation.1 -> annotation.IDLE.eid -> SKIP )
)
( chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.5 -> DC
  []
  chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.0 -> DA
  []
  chan_PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET.1 -> DB
 )
)

```

310
The specification covered in Figure 168 can be perceived as a hybrid combination of events modelling low level hardware and its higher level conceptual meanings. The purpose of this model is to provide an avenue to link the model of the segment of logic circuit (when driven correctly), to the desired annotation only model describing conceptually the task it is trying to perform. Connecting these two models together is achieved through linking both of them to the hybrid model through the use of assertions and the process to do so it is broken down into three sections. The first part involves demonstrating that the annotations contained within the hybrid model behaves identically to that of the higher level full annotation only model so long as the low level events are hidden (see Figure 169).
Figure 169: Hybrid annotations behave identically to annotation only specification

The second part involves demonstrating that the outer level annotations from the annotation only model (having the annotations for the internal components hidden), behaves within the allowed behaviour for a generic control flow process (see Figure 170).
Appendix J: Pattern Matched Selective Renaming Alternative

```plaintext
SYSTEM_INTERFACE = { chan0, chan10, chan2 }
APROCO = PROC_BOOLEAN_HIGHER OUTER_SPEC_NORESET(1)
APROC1 = PROC_PROCESS_HIGHER OUTER_SPEC(2)
APROC2 = PROC_PROCESS_HIGHER OUTER_SPEC(3)

ANNOTATION ONLY_SPEC = PROC_PROCESS_IP_PULL_SPEC_NORESET(0, APROC0, 1, APROC1, 2, APROC2, 3)
OUTER_ANNOTATION ONLY_SPEC = ANNOTATION ONLY_SPEC \ {
  annotation.IDLE.1,
  annotation.BOOLEANREAD.1,
  annotation.BOOLEANREADALLOWED.1,
  annotation.BOOLEANFALSE.1,
  annotation.BOOLEANTRUE.1,
  annotation.IDLE.2,
  annotation.START.2,
  annotation.NOTFINISHED.2,
  annotation.IDLE.3,
  annotation.START.3,
  annotation.NOTFINISHED.3,
  annotation.FINISH.3
}

assert PROC_PROCESS_HIGHER OUTER_SPEC(0) \ [PD= OUTER_ANNOTATION ONLY_SPEC]
```

Figure 170: Check if Outer Annotation Behaviour from Hybrid Model is Allowed

The third part involves demonstrating that the hybrid model with the annotation events renamed back to the corresponding low level hardware events, behaves identically to the correctly driven circuit (see Figure 171).
Appendix J: Pattern Matched Selective Renaming Alternative

```plaintext
SYSTEM_INTERFACE = [ chan0, chan10, chan2 ]
SYSTEM_CONTROL = PROC_PROCESS_CONTROL_NORESET(chan0, chan2, chan10)

HPROC0 = PROC_BOOLEAN_GENERIC_HYBRID_SPEC_NORESET(
  chan0, chan11, chan3, chan4, 1)
HPROC1 = PROC_PROCESS_GENERIC_HYBRID_SPEC_NORESET(
  chan0, chan5, chan6, 2)
HPROC2 = PROC_PROCESS_GENERIC_HYBRID_SPEC_NORESET(
  chan0, chan7, chan8, 3)
APROC0 = PROC_BOOLEAN_HIGHER_OUTER_SPEC_NORESET(1)
APROC1 = PROC_PROCESS_HIGHER_OUTER_SPEC(2)
APROC2 = PROC_PROCESS_HIGHER_OUTER_SPEC(3)

HYBRID_SPEC = PROC_PROCESS_IF_HYBRID_FULL_SPEC_NORESET(
  chan0, chan2, chan10, 0,
  HPROC0, chan11, chan3, chan4, 1,
  HPROC1, chan5, chan6, 2,
  HPROC2, chan7, chan8, 3,
  chan9
)

HYBRID_SPEC_RENAMED =
  ( HYBRID_SPEC
    ([ annotation.IDLE.0 <- chan2.0,
     annotation.START.0 <- chan2.1,
     annotation.NOTFINISHED.0 <- chan10.0,
     annotation.FINISH.0 <- chan10.1,
     annotation.IDLE.1 <- chan11.0,
     annotation.BOOLEANREAD.1 <- chan11.1,
     annotation.BOOLEANREADALLOWED.1 <- chan3.0,
     annotation.BOOLEANFALSE.1 <- chan4.0,
     annotation.BOOLEANTRUE.1 <- chan4.1,
     annotation.IDLE.2 <- chan5.0,
     annotation.START.2 <- chan5.1,
     annotation.NOTFINISHED.2 <- chan6.0,
     annotation.FINISH.2 <- chan6.1,
     annotation.IDLE.3 <- chan7.0,
     annotation.START.3 <- chan7.1,
     annotation.NOTFINISHED.3 <- chan8.0,
     annotation.FINISH.3 <- chan8.1
   ])
  )

FULL_SYSTEM_WITH_CONTROL =
  ( SYSTEM_WITH_INTERNALS [ SYSTEM_INTERFACE ] SYSTEM_CONTROL )

assert FULL_SYSTEM_WITH_CONTROL [PD= HYBRID_SPEC_RENAMED
assert HYBRID_SPEC_RENAMED [PD= FULL_SYSTEM_WITH_CONTROL

Figure 171: Hybrid model with annotations renamed behaves identically to correctly driven circuit

Ideally the process of linking the hybrid model back to the low level implementation model would not be needed to be performed. If pattern matched selective renaming was allowed (possibly through expanding CSP to combine regular expressions to pattern match and select which events a renaming is applied to), it would be possible to automatically generate the hybrid model from the segment of logic circuit modelled, thus guaranteeing that the hybrid model derives its behaviour from the implementation being examined.

314
Appendix J: Pattern Matched Selective Renaming Alternative

Each individual component is modelled and run in parallel.

Constrain input signals to disallow incorrect outer level driving.

Implementation Hardware Specification

Specification modified by hand, changing specific low level events to annotation events

Hybrid Specification

Hide low level events

Implementation Clock Cycle Software Specification

Implementation Software Specification

Equivalence

Selectivity pattern matched event renaming

CSP Model of Net List disallowing incorrect driving

CSP Model of Net List

Equivalence

Refinement of

Super Type Hardware Specification

Hybrid Specification with annotations renamed as low level events

Hybrid Specification

Refinement of

Software Model

Software Model

Refinement of

Super Type Clock Cycle Software

Super Type Software Specification

Figure 172: Overview of proof structure for pattern matched selective renaming alternative
Figure 172 illustrates diagrammatically for this section of work, the structure of how the CSP models representing the hardware at various levels of abstraction are used in the proof, and relate to each other. The section highlighted in yellow is the area of the proof structure that differs from that used in the main body of the work (chapter 4 and Appendix D to Appendix G), along with the slightly modified method illustrated in Appendix H and Appendix I. This yellow section is a simplification of the proof structure, and was presented at the cost of the robustness of the model (i.e. not checking the reset functionality), in an attempt to simplify the presentation of the structure that the proof framework uses to prove individual grammatical constructs.

### J.2 Conclusion

It is through the combination of the proofs that integrate the various models together that a segment of logic circuit can be demonstrated to be a valid implementation. This demonstration of the validity of a components implementation, combined with a proof of composition (achieved through proving implementations are a refinement of a super type specification and so can be replaced wherever that super type specification was used), provides an avenue to prove the process of generating the complete logic circuit (without the need to model the complete circuit).
Appendix K  Simplified CSP Models
Without Reset Feature

This section contains the modified models for the proof of the "IF THEN ELSE" component that has the reset functionality removed (see Appendix J).

K.1 Logic Components

The removal of the reset functionality from the models has no effect on the combinatorial logic (e.g. AND gates, OR gates, NOT gates, etc), but requires a slight adaptation to the clocked logic components (see Figure 173). The D-Type Flip Flop is modelled in an OI-SEQ manner.

```plaintext
-- Instantiations of a D_TYPE_FLIP_FLOP calls this process and provides channels
-- used for its inputs and outputs as arguments. 'q_out' is a single channel to
-- output to. 'd_in' is a single channel to input from. 'clock' is
-- an event representing a low-to-high clock transition.
D_TYPE_FLIP_FLOP_NORESET(clock, d_in, q_out) =
  let
    -- Check if the output is connected to the input.
    A =
      inter( [ q_out |], [ d_in |] ) == {} & B(0)
       
      inter( [ q_out |], [ d_in |] ) != {} & STOP

    B(x) =
      q_out ! x -> d_in ? y -> clock ? 1 -> B(y)
  within A
```

*Figure 173: D-Type Flip Flop with no Reset function*

K.2 Super Type Specifications

K.2.1 Control Flow Process

This section contains the modified models that specify a super type control flow process that the proof in Appendix J builds upon.
K.2.1.1 Control Flow Process: Correct Control

This CSP model (see Figure 174) is used to limit a process so that it can only accept possible valid input signals, this is to enable implemented sub-type components to have the outer layer of their logic correctly driven when performing the checks and proofs. The aim for this is to check an implemented component holds true to the assumption that so long as it is driven correctly, it will correctly drive any internal components.

```
PROC_PROCESS_CONTROLL_NORESET(clock, start, finish) =
let
  A =
    start?x ->
    ( x == 0 & clock?1 -> finish!0 -> A 
         []
         x == 1 & B
    )

  B = clock?1 ->
    ( finish!0 -> start?0 -> B 
       []
       finish!1 -> A
    )
within finish!0 -> A
```

Figure 174: Generic Control Flow Specification - Correct Driving Limiter

K.2.1.2 Control Flow Process: Low Level Specification

This model (see Figure 175) specifies all the valid and allowable low level behaviour of this type of super type component. The purpose is to describe the interface boundary behaviours, thus enabling implemented components to refinement check against it proving there behaviours are within the requirements for it to be a sub-type of this super-type.

```
PROC_PROCESS_DESIRED_GENERIC_SPEC_NORESET(clock, start, finish) =
let
  A =
    start?x -> clock?1 ->
    ( x == 0 & finish!0 -> A 
       []
       x == 1 &
       ( internalChoice -> finish!1 -> A 
          [] -- |-
          internalChoice -> finish!0 -> B
       )
    )

  B = start?0 -> clock?1 ->
    ( internalChoice -> finish!1 -> A 
       [] -- |-
       internalChoice -> finish!0 -> B
    )
within finish!0 -> A
```

Figure 175: Low Level Generic Control Flow Specification
Appendix K: Simplified CSP Models Without Reset Feature

K.2.1.3 Control Flow Process: Low Level Specification With Deadlocking

This CSP model (see Figure 176) is based on the one covered in section K.2.1.2, but altered so that it also accepts invalid driving input signals submitted to it. These invalid input driving signals are followed by an explicitly defined ‘STOP’ that will deadlock the model should it ever be reached. Similar to the specification in section K.2.1.2, the returned output signals will be all possible valid permutations allowed (internal choice is utilised to create those permutations, so long as it is driven correctly).

The reason why this model will accept invalid driving signals is to enable the model of any component connected to it the opportunity to provide any driving signals it may choose, this process will not limit or remove the possibility for the other component models to provide invalid signals to this one as an option when they are run in alphabetised parallel. The purpose of this is to enable possibility to check that if this specification is used as an internal component, so long as the outer component is driven correctly, this component will be driven correctly.

PROC_PROCESS_GENERIC_SPEC_NORESET(clock, start, finish) =

let

A =

start? x -> clock? l ->

( x == 0 & finish!0 -> A
[ ]

x == 1 &

( internalChoice -> finish!1 -> A
[ ] -- [ ]

internalChoice -> finish!0 -> B )
)

B =

start? 1 -> STOP
[ ]

start? 0 -> clock? l ->

( internalChoice -> finish!1 -> A
[ ] -- [ ]

internalChoice -> finish!0 -> B )

within finish!0 -> A

Figure 176: Low Level Generic Control Flow Specification with Explicit Deadlocking

K.2.1.4 Control Flow Process: High Level Specification

This CSP model (see Figure 177) is an annotation only clock cycle based higher conceptual specification. It is used as a comparison for the extracted annotations from the annotated low level hardware models. The model is sufficiently small so that it is unlikely
that 'chase' compression should be needed to be applied, which is why internal choice (i.e. '|-|') is used instead of using an extra event to simulate internal choice.

```plaintext
PROC_PROCESS_HIGHER_OUTER_SPEC(id) =
let
A =
  annotation.START.id -> B
|[]
  annotation.IDLE.id -> A
B =
  annotation.NOTFINISHED.id -> B
|-
  annotation.FINISH.id -> A
within A
```

Figure 177: Generic Control Flow Annotation Specification

K.2.2 Boolean Process

This section contains only two of the modified models that would be required to fully specify a Boolean super type specification, this is due to the fact that the proof covered in Appendix J only utilises these two models as internal component place fillers.

K.2.2.1 Boolean Process: Low Level Specification With Deadlocking

This CSP model (see Figure 178) is the low level behaviour altered so that it also accepts invalid driving input signals submitted to it. These invalid input driving signals are followed by an explicitly defined 'STOP' that will deadlock the model should it ever be reached. The specification returns all possible valid permutations of allowed output signals so long as it is driven correctly, internal choice is utilised to create those permutations.

The reason why this model will accept invalid driving signals is to enable the model of any component connected to it the opportunity to provide any driving signals it may choose, this process will not limit or remove the possibility for the other component models to provide invalid signals to this one as an option when they are run in alphabetised parallel. The purpose of this is to enable possibility to check that if this specification is used as an internal component, so long as the outer component is driven correctly, this component will be driven correctly.
Appendix K: Simplified CSP Models Without Reset Feature

PROC BOOLEAN ßBNBRIC_SPEC NORESET(clock, bool_read, bool_readallowed, state) =
let
A =
  bool_readallowed!0 -> state!0 -> B
B =
  bool_read?x -> clock?1 ->
    ( x == 0 & A
     []
     x == 1 & D
    )
D =
  internalChoice -> bool_readallowed!0 -> state!0 -> E
  [] -- |&|
  internalChoice -> bool_readallowed!1 ->
    ( internalChoice -> state!0 -> B
     [] -- |&|
     internalChoice -> state!1 -> B
    )
E =
  bool_read?1 -> STOP
  []
  bool_read?0 -> clock?1 -> D
within A

Figure 178: Low Level Boolean Specification with Explicit Deadlocking

K.2.2.2 Boolean Process: High Level Specification

This CSP model (see Figure 179) is an annotation only clock cycle based higher conceptual specification. It is used as a comparison for the extracted annotations from the annotated low level hardware models. The model is sufficiently small so that it is unlikely that ‘chase’ compression should be needed to be applied, which is why internal choice (i.e. ‘|&|’) is used instead of using an extra event to simulate internal choice.

PROC_BOOLEAN_HIGHER_OUTER_SPEC_NORESET(id) =
let
A =
  annotation.IDLE.id -> A
  []
  annotation.BOOLEANREAD.id -> B
B =
  annotation.BOOLEANDONTREAD.id -> B
  []
  ( annotation.BOOLEANREADALLOWED.id ->
    ( annotation.BOOLEANFALSE.id -> A
     []
     annotation.BOOLEANTRUE.id -> A
    )
  )
within A

Figure 179: Generic Control Flow Annotation Specification
K.3 **Implemented “IF THEN ELSE” Component**

This section contains modified models from the “IF THEN ELSE” component, with the reset functionality removed.

**K.3.1 IF THEN ELSE: Low Level Specification**

This CSP model (see Figure 180) which is similar to the model defined in section K.2.1.2, specifies the valid and allowable low level behaviour that this implemented component may perform at its outer boundary. It may utilises internal choice to determine the possible output behaviour it can perform, although it is not a requirement (e.g. boolean true, boolean false, SKIP, STOP, all have well defined fixed behaviours that do not rely on other internal components). It is useful to note that some implemented components may have the allowable interface boundary behaviour that is identical to that of its generic super type component (e.g. boolean comparisons, PAR), where as other components will have an interface boundary behaviour that is a refinement of its super type component (e.g. boolean true, boolean false, SEQ).

```plaintext
PROC_PROCESS_IF_DESIRED_SPEC_NORESET(clock, start, finish) =
let
  A =
    start?0 -> clock?1 -> finish!0 -> A
  []
    start?1 -> clock?1 -> finish!0 -> B
  B =
    start?0 -> clock?1 ->
      (internalChoice -> finish!1 -> A
        [] -- [\]
        internalChoice -> finish!0 -> B
      )
within finish!0 -> A
```

*Figure 180: Low Level 'IF' Component Desired Specification*

**K.3.2 IF THEN ELSE: Low Level Specification With Deadlocking**

This CSP model (see Figure 181) is similar to the model covered in section K.2.1.3, the CSP model is the model covered in section K.3.1 but altered so that it will accept incorrectly driven input signals followed by explicitly defined deadlocking (i.e. STOP).
Appendix K: Simplified CSP Models Without Reset Feature

PROC_PROCESS_IF_GENERIC_SPEC_NORESET(clock, start, finish) =
let
  A =
    start?x -> clock?1 -> finish!0 ->
    ( x == 0 & A
    []
    x == 1 & B )
  B =
    start?1 -> STOP
    []
    start?0 -> clock?1 ->
    ( internalChoice -> finish!1 -> A
    [] -- |-
    internalChoice -> finish!0 -> B )
within finish!0 -> A

Figure 181: Low Level 'IF' Component Generic Specification with Explicit Deadlocking

K.3.3 IF THEN ELSE: High Level Specification

This CSP model (see Figure 182), is the expected behaviour of the 'IF' component from an annotation only perspective, with annotation models of the internal component (i.e. the boolean condition, the 'then' process and the 'else' process) having to be supplied. If the internal annotation components supplied are the corresponding generic specifications, the CSP model will demonstrate all the possible behaviours of the 'IF' component, describing both how driving the component drives the internal components and the internal components behaviour effects the outputs of this component. The reason why this model was designed to take in processes representing the internal components is so that if the supplied internal component specifications are a refinement of the corresponding generic specification, they will limit the behaviour dictated in the 'IF' component to that which describes what should conceptually occur in the hardware.

Figure 182: IF Component Annotation Only Specification

channel chan_PROC_PROCESS_IF_FULL_SPEC_NORESET : {0..5}
PROC_PROCESS_IF_FULL_SPEC_NORESET(
  id, boolproc, bid, thenproc, tid, eleeproc, eid
) =
let
  -- Perform checks on the input parameters
  A =
    inter( { id }, { bid, eid, tid } ) == {} & B
    []
    inter( { id }, { bid, eid, tid } ) != {} & STOP
  -- Perform the model
  -- Run the internal components in parallel with the remainder of the IF
  -- component
  B =
Appendix K: Simplified CSP Models Without Reset Feature

let DA =

( ( annotation.IDLE.tid -> SKIP )

( annotation.IDLE.eid -> SKIP )

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.0 -> DA

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.1 -> DB )

DB =

annotation.BOOLEANREAD.bid ->

( ( ( annotation.IDLE.tid -> SKIP )

( annotation.IDLE.eid -> SKIP )

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.2 -> DB )

[]

annotation.BOOLEANREAD.bid ->

( ( ( annotation.IDLE.tid -> SKIP )

( annotation.START.eid -> SKIP )

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.3 -> DC )

[]

annotation.BOOLEANREAD.bid ->

( ( ( annotation.START.tid -> SKIP )

( annotation.IDLE.eid -> SKIP )

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.4 -> DD )

[]

annotation.BOOLEANTRUE.bid ->

( ( ( annotation.START.tid -> SKIP )

( annotation.IDLE.eid -> SKIP )

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.5 -> DC )

[]

[[]

annotation.IDLE.tid ->

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.5 -> DC

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.0 -> DA

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.1 -> DB )

DD =

annotation.IDLE.tid ->

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.5 -> DC

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.0 -> DA

( chan_PROC_PROCESS_IF_FULL_SPEC_NORESET.1 -> DB )

within DA

--
Appendix K: Simplified CSP Models Without Reset Feature

\[
E = \\
\{ C \mid \{ \text{chan\_PROC\_PROCESS\_IF\_FULL\_SPEC\_NORESET} \} \} \\
\} \setminus \{ \text{chan\_PROC\_PROCESS\_IF\_FULL\_SPEC\_NORESET} \} \\
\text{within A}
\]
Appendix L Expected Run Time

To determine the timings for expected output of a software application that has been converted into hardware, the following rules stated in this section can be applied while stepping through an instance of the application.

L.1 Generic: Control Flow Process

This super type specification for control flow process is allowed to take one or more clock cycles to complete.

L.1.1 Implemented: SKIP (Delay)

This component takes one clock cycle to complete.

L.1.2 Implemented: STOP (Deadlock)

This component never completes.

L.1.3 Implemented: SEQ

This component takes the sum total of the time that all its internal processes take to complete, i.e. it completes after the all its internal processes have completed running sequentially.

L.1.4 Implemented: PAR

This component takes the same as the longest of its internal components take to complete, i.e. it completes when all its internal components have completed with them running in parallel.

L.1.5 Implemented: Assignment

This component takes the sum total of the time its internal 'Read' and 'Store' components take to complete (see sections L.3 and L.4 respectively), i.e. it completes after the read and store have sequentially finished (in that order).
L.1.6  **Implemented: Channel Declaration**

This component takes one clock cycle to complete.

L.1.7  **Implemented: Variable Declaration**

This component takes the storage time for the variable (see section L.5), to complete.

L.1.8  **Implemented: IF (IF THEN ELSE)**

This component takes the summation of the length of time for its internal 'Boolean Process' (see section 0) and its corresponding 'THEN' or 'ELSE' process (dependent on if the test returns true or false respectively).

L.1.9  **Implemented: While**

This component's time to completion is dependent on the specific result of its internal 'Boolean Process' component each time the loop is triggered, combined with the length of its internal 'Control Flow Process' if result of the test was true, see Figure 183.

![Figure 183: How to compute the completion time for a 'While' component](image)

L.1.10  **Implementation: Output Channel (Sending Along a Channel)**

This component takes the summation of the length of time for its internal 'Read' component, along with any channel communication delay. The channel communication delay is comprised of any delay due to the need to wait for the other side of the channel to
be ready, and also any delay for communication across the channel medium. This channel communication delay will be explained further in section L.6.

L.1.11 Implementation: Input Channel (Reading from a Channel)

This component takes the summation of the length of time for any channel communication delay, along with the delay due to its internal 'Store' component. The channel communication delay is comprised of any delay due to the need to wait for the other side of the channel to be ready, and also any delay for communication across the channel medium. This channel communication delay will be explained further in section L.6.

L.2 Generic: Boolean Process

This super type specification for a 'Boolean Process' is allowed to take one or more clock cycles to complete.

L.2.1 Implemented: True

This component takes one clock cycle to complete.

L.2.2 Implemented: False

This component takes never completes.

L.2.3 Implementation: Boolean AND

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.

L.2.4 Implementation: Boolean OR

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.

L.2.5 Implementation: EQUAL

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.
L.2.6 Implementation: NOT EQUAL

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.

L.2.7 Implementation: LESS THAN

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.

L.2.8 Implementation: GREATER THAN

This component takes the same as the longest of its internal 'Boolean Process' components take to complete.

L.3 Generic: Read

This super type specification for a 'Read' is allowed to take one or more clock cycles to complete. Specific implementations of variables can have differing times to completion and allows for implementation of off chip storage of variables, which might take multiple clock cycles to access.

L.3.1 Implementation: UINT Constant

This component takes a single clock cycle to complete.

L.3.2 Implementation: Read for FlipFlopStorage Variable

This component takes a single clock cycle to complete.

L.3.3 Implementation: Read for WatchedFlipFlopStorage Variable

This component takes a single clock cycle to complete.

L.4 Generic: Store

This super type specification for a 'Store' is allowed to take one or more clock cycles to complete. Specific implementations of variables can have differing times to completion
and allows for implementation of off chip storage of variables, which might take multiple clock cycles to set.

L.4.1 Implementation: Store for FlipFlopStorage Variable

This component takes a single clock cycle to complete.

L.4.2 Implementation: Store for WatchedFlipFlopStorage Variable

This component takes a single clock cycle to complete.

L.5 Generic: Variables

The current implemented variables are all on chip and are achieved by utilising flip flops to keep the data (e.g. FlipFlopStorage component). The 'WatchedFlipFlopStorage' component is identical to that of the 'FlipFlopStorage' component, but it has the values from the flip flops used to store the data values connected to output pads so that the values can be examined from outside the circuit.

Sections L.3.2 and L.3.3 specify the 'Read' interfaces for the implemented variables respectfully (i.e. for the 'FlipFlopStorage' and 'WatchedFlipFlopStorage' components), whereas sections L.4.1 and L.4.2 specify the 'Store' interfaces.

L.6 Generic: Channels

Currently I have implemented three different channels, these are 'Channel_NotPlaced', 'Channel_PlacedInput' and 'Channel_PlacedOutput'. These three types of channels are basically all the same, apart from the placed input channel having its 'Channel Output' placed to the outside of the chip, and the placed output channel having its 'Channel Input' placed to the outside of the chip, thus allowing data to be supplied or extracted respectfully from or to the circuit via channels.

The medium that the current implemented channels use is digital logic (i.e. it does not communicate over a network, etc). Because of this, the communication delay stated in sections L.1.10 and L.1.11 is 0. Examples for the timings that two ends of a channel have on the completion timings of each other is demonstrated in Figure 184, Figure 185 and Figure 187, where as increasing the communication delay (i.e. the number of clock cycles for information to pass from one side of the channel to the other) would impact and increase the completion time. It should be noted that the logic that logic that represents the channel communication has not been optimised fully, although it still is a
viable implementation. This can be seen in Figure 184, Figure 185 and Figure 187, by the channel output taking one extra clock cycle to terminate than the start of the channel input.

Figure 184: Sending and receiving started at the same time on a 0 communication delay channel

Figure 185: Sending started 1 cycle before receiving on a 0 communication delay channel
Appendix L: Expected Run Time

Figure 186: Sending started 2 cycles before receiving on a 0 communication delay channel

Figure 187: Reading started 1 clock cycle before sending on a 0 communication delay channel
Appendix M  Esterel SCADE 6.0 - Design Verifier Problem

This appendix provides a simple example illustrating the inconsistency of the SCADE design verifier’s ability to analyse a model, even though the SCADE tool can compile and simulate the model. SCADE is a model based development environment dedicated to safety critical embedded software, produced by Esterel Technologies (http://www.esterel-technologies.com).

M.1 A Simple Model Using Enumerated Types

The model illustrated in Figure 188 (that utilises the enumerated types defined in Table 4), is a simple example. Two inputs (of different types) are taken as inputs and compared against constants, these results are then passed to an AND gate, whereby the result is passed as the output.

Table 4: Definitions of enumerated type used in Figure 188

<table>
<thead>
<tr>
<th>Name of Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyEnumeratedType1</td>
<td>enum {EN1_A, EN1_B}</td>
</tr>
<tr>
<td>MyEnumeratedType2</td>
<td>enum {EN2_A, EN2_B}</td>
</tr>
</tbody>
</table>
M.2 Breaking the Design Verifier

Through a simple modification of the model illustrated in Figure 188, whereby ‘Input2’ is replaced with a constant (see Figure 189), one can produce a model that the design verifier can not analyse. Placing a ‘proof objective’ on the output and attempting to analyse the model results in the design verifier producing a “Not_found” error. It should be noted however that the tool can compile and simulate the model.

![Figure 189: Modifying Figure 188 to break the design verifier](image)

M.3 Trying to Locate the Problem

Having identified a model whereby the SCADE tool can compile and simulate it, but the design verifier can not analyse it, one can then break the model down into simpler segments in an attempt to identify the section that is causing the error.

Replacing one subsection of the model with a ‘true’ constant, we get the model illustrated in Figure 190. When a ‘proof objective’ is placed on the output, the design verifier can analyse the model. Although this may appear to suggest that the problem is located in the section of the model that was removed, this is not the case.

![Figure 190: Simplifying the model - Version 1](image)

Through replacing an alternative section of the model with a ‘true’ constant, we get the model illustrated in Figure 191. When a ‘proof objective’ is placed on the output, the design verifier can analyse the model. This could be interpreted as suggesting that the problem is located not in the first part that was removed but the second.
M.4 Conclusion

With the two different models (Figure 190 and Figure 191) suggesting the cause of the problem exists in different sections of the model, which, if any of these theories is correct? Without the ability to examine the source code of the SCADE tool and the integrated Prover Plug-In [Sheeran, 2000] that underpins its design verifier [Köhler and Kant, 2004], it is impossible to specify the exact cause of the bug. Even though this is the case, it is possible to reason about the construction of the tool, thus identifying possible causes or areas of concern.

With the SCADE tool being able to compile and simulate the models (Figure 188 to Figure 191), this suggests that the problem is not located in the internal data structure that represents the model. If one assumes that Stålmarck’s method of tautology checking using propositional logic ([Harrison, 1996] and [Sheeran & Stålmarck, 2000]) that underpins the Prover Plug-In [Sheeran, 2000] is sound, one is lead to believe that the location of the error or bug is contained within how the tool was developed.

As the Prover Plug-In was developed by a third party (the product is utilised by several companies and tools), a sensible assumption would be that it contains its own internal data structures. It is highly unlikely that the SCADE compiler utilises this data structure as its internal representation, a theory that is supported by the fact that the SCADE compiler was designed and developed independently of the Prover Plug-In. This is clearly illustrated by the fact that [Köhler and Kant, 2004] describe the augmentation of the SCADE compiler with the formal proof tool. So, under the assumption that there are at least two internal data structures that represent the model, a process must exist to translate the SCADE internal data structure into one that the Prover Plug-In formal verification tool can utilise. I believe that this is the most likely location for the error or bug, but as previously indicated, I can not ascertain as to whether this is correct, without access to the SCADE source code.
Appendix N  OCCAM: Digital Clock

SEQ
  PLACED UINT1 sfd AT "SF_D":
  sfd := 1
CHAN OF UINT1 chanLCDRS:
CHAN OF UINT4 chanSFD:
CHAN OF UINT1 startDelay:
CHAN OF UINT1 delayFinished:
PAR
  -- LCD_Nibble
  -- When a command is received on the chanLCDRS and chanSFD channels,
  -- send the corresponding command, correctly timed, to the LCD controller
SEQ
  PLACED UINT4 sfd AT "SFD_8", "SFD_9", "SFD_10", "SFD_11":
  PLACED UINT1 lcode AT "LCD_E":
  PLACED UINT1 lcdrs AT "LCD_RS":
  PLACED UINT1 lcdrw AT "LCD_RW":
  UINT2 holdcom:
  UINT5 comdelay:
  WHILE TRUE
    SEQ
      PAR
      comdelay := 0
      holdcom := 0
      chanLCDRS ? lcdrs
      chanSFD ? sfd
      SKIP
      lcode := 1
      WHILE holdcom <> 3
        holdcom := holdcom PLUS 1
      SKIP
      lcode := 0
      SKIP
      WHILE comdelay <> 16
        comdelay := comdelay PLUS 1
      SKIP
      -- EndOf: LCD_Nibble
      -- LCD_Delay
      -- Creates a delay of the correct length between the parts of a nibbled
      -- command
      SEQ
        UINT1 junk:
        UINT11 waitcom:
        WHILE TRUE
          SEQ
            startDelay ? junk
            WHILE waitcom <> 666
              waitcom := waitcom PLUS 1
            PAR
              waitcom := 0
              delayFinished ? 1
            SKIP
            -- EndOf: LCD_Delay
            -- Main
            -- Runs the LCD Power on initialisation, followed by the main body of the
            -- application
            SEQ
              -- LCD_PowerOnInit
              SEQ
                UINT1 junk:
                UINT7 initWait:
                WHILE initWait <> 103
SEQ
startDelay ! 1
initWait := initWait PLUS 1
delayFinished ? junk

PAR
chanLCDRS ! 0
chanSFD ! 3

SEQ
startDelay ! 1
delayFinished ? junk
startDelay ! 1
delayFinished ? junk
startDelay ! 1
delayFinished ? junk

PAR
chanLCDRS ! 0
chanSFD ! 2

SEQ
startDelay ! 1
delayFinished ? junk

SEQ
chanLCDRS ! 0
chanLCDRS ! 0

SEQ
chanSFD ! 2
chanSFD ! 8

SEQ
startDelay ! 1
delayFinished ? junk

PAR
SEQ
chanLCDRS ! 0
chanLCDRS ! 0

SEQ
chanSFD ! 0
chanSFD ! 6

SEQ
startDelay ! 1
delayFinished ? junk

PAR
SEQ
chanLCDRS ! 0
chanLCDRS ! 0

SEQ
chanSFD ! 0
chanSFD ! 14

SEQ
startDelay ! 1
delayFinished ? junk

PAR
SEQ
chanLCDRS ! 0
chanLCDRS ! 0

SEQ
chanSFD ! 0
chanSFD ! 1

WHILE initWait <> 41
SEQ
startDelay ! 1
initWait := initWait PLUS 1
delayFinished ? junk
initWait := 0
-- EndOf: LCD_PowerOnInit
SEQ
CHAN OF UINT4 chanTimeH1:
CHAN OF UINT4 chanTimeH0:
CHAN OF UINT4 chanTimeM1:
CHAN OF UINT4 chanTimeH:,
CHAN OF UINT4 chanTimeM:,
CHAN OF UINT4 chanTimeS:,
CHAN OF UINT1 buttonState:
CHAN OF UINT1 button2State:
PAR
  -- DigitalClock
SEQ
  UINT4 timeH:
  UINT4 timeM:
  UINT4 timeS:
  UINT1 incH:
  UINT1 incM:
  UINT23 counter:
WHILE TRUE
  SEQ
    SEQ
      PAR
        chanTimeH ! timeH
        chanTimeM ! timeM
        chanTimeS ! timeS
        buttonState ? incH
        button2State ? incM
        WHILE counter <> 8333331
          counter := counter PLUS 1
        PAR
        counter := 0
        IF
          incH = 1
            TRUE
             SEQ
              SKIP
              SKIP
            SEQ
            SKIP
            SKIP
          IF
          incM = 1
            TRUE
              SEQ
              SKIP
              SKIP
            PAR
            IF
              timeH = 0
                IF
                  timeM = 10
                    TRUE
                      SEQ
                      SKIP
                      SKIP
                    TRUE
                      IF
                        timeH = 3
                          TRUE
                            SEQ
                            SKIP
                            SKIP
                        SEQ
                        SKIP
                      SEQ
                    SEQ
                  TRUE
                SEQ
              SEQ
            SEQ
          SEQ
        SEQ
      SEQ
      PAR
      counter := 0
    SEQ
  SEQ
  PAR
  -- DigitalClock
SEQ
SEQ
SKIP
SKIP
IF
    timeM0 = 10
PAR
    timeM0 := 0
    timeM1 := timeM1 PLUS 1
TRUE
SEQ
    SKIP
SEQ
IF
    timeM1 = 6
    timeM1 := 0
TRUE
SEQ
    SKIP
SEQ
PAR
    chanTimeH1 := timeH1
    chanTimeH0 := timeH0
    chanTimeM1 := timeM1
    chanTimeM0 := timeM0
    chanTimeS1 := timeS1
    chanTimeS0 := timeS0
    button1State ? incH
    button2State ? incM
WHILE counter <> 8333326
    counter := counter PLUS 1
PAR
    counter := 0
    timeS0 := timeS0 PLUS 1
PAR
    IF
        incH = 1
        timeH0 := timeH0 PLUS 1
    TRUE
    SEQ
        SKIP
        SKIP
    IF
        incM = 1
        timeM0 := timeM0 PLUS 1
    TRUE
    SEQ
        SKIP
        SKIP
    IF
        timeS0 = 10
    PAR
        timeS0 := 0
        timeS1 := timeS1 PLUS 1
    TRUE
    SEQ
        SKIP
SEQ
IF
    timeS1 = 6
    PAR
        timeS1 := 0
        timeM0 := timeM0 PLUS 1
    TRUE
    SEQ
        SKIP
SEQ
IF
timeM0 = 10 OR timeM0 = 11
PAR
  timeM0 := timeM0 MINUS 10
  timeM1 := timeM1 PLUS 1
TRUE
SEQ
  SKIP
  SKIP
IF
timeM1 = 6
PAR
  timeM1 := 0
  timeH0 := timeH0 PLUS 1
TRUE
SEQ
  SKIP
  SKIP
IF
timeM1 = 0
IF
timeH0 = 10 OR timeH0 = 11
PAR
  timeH0 := timeH0 MINUS 10
  timeH1 := timeH1 PLUS 1
TRUE
SEQ
  SKIP
  SKIP
TRUE
IF
timeH0 = 3 OR timeH0 = 4
PAR
  timeH0 := timeH0 MINUS 2
  timeH1 := 0
TRUE
SEQ
  SKIP
  SKIP
-- EndOf: DigitalClock
-- DebounceHour
-- Debounce and sample the input for increasing the hours
SEQ
  PLACED UINT1 button ON "inch":
  UINT1 buttonSampled:
  PLACED UINT1 currentState AT "button":
  UINT1 nextState:
  UINT1 stable:
  UINT22 count:
PAR
  stable := 1
  buttonSampled := button
WHILE TRUE
SEQ
  PAR
    button1State := currentState
    WHILE count <> 3124999
      PAR
        buttonSampled := button
        count := count PLUS 1
        IF
          nextState <> buttonSampled
          stable := 0
        TRUE
        SEQ
          SKIP
          SKIP
      PAR
        count := 0
Appendix N: OCCAM: Digital Clock

```occam
stable := 1
IF
  stable = 1
  currentState := nextState
  TRUE
  currentState := 0
SEQ
  SKIP
nextState := buttonSampled

-- EndOf: DebounceHour
-- DebounceMin
-- Debounce and sample the input for increasing the minutes
SEQ
PLACED UINT1 button ON "incM":
UINT1 buttonSampled:
PLACED UINT1 currentState AT "buttonM":
UINT1 nextState:
UINT1 stable:
UINT22 count:
PAR
  stable := 1
  buttonSampled := button
WHILE TRUE
SEQ
  PAR
  nextState := currentState
  WHILE count <> 3124999
  PAR
    buttonSampled := button
    count := count PLUS 1
    IF
      nextState <> buttonSampled
      stable := 0
      TRUE
    SEQ
    SKIP
    SKIP
  PAR
    count := 0
    stable := 1
    IF
      stable = 1
      currentState := nextState
      TRUE
      currentState := 0
    SEQ
    SKIP
    nextState := buttonSampled

-- EndOf: DebounceMin
-- LCD_DisplayTime
SEQ
UINT4 timeH1:
UINT4 timeHO:
UINT4 timeM1:
UINT4 timeMO:
UINT4 timeS1:
UINT4 timeSO:
UINT1 junk:
UINT15 counter:
WHILE TRUE
SEQ
  PAR
    counter := 0
    chanTimeH1 ? timeH1
    chanTimeH0 ? timeH0
    chanTimeM1 ? timeM1
    chanTimeM0 ? timeM0
    chanTimeS1 ? timeS1
```

342
Appendix N: OCCAM: Digital Clock

chanTimeS0 ? timeS0
SEQ
PAR
SEQ
chanSFD 1 3
chanSFD 1 timeH1
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 timeH0
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 10
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 timeM1
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 timeM0
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 timeS1
SEQ
chanLCDRS 1 1
chanLCDRS 1 1
startDelay 1 1
delayFinished ? junk
PAR
SEQ
chanSFD 1 3
chanSFD 1 3

343
chanSFD ! timeS0
SEQ
chanLCDRS ! 1
chanLCDRS ! 1
startDelay ! 1
delayFinished ? junk
PAR
SEQ
chanLCDRS ! 0
chanLCDRS ! 0
SEQ
chanSFD ! 0
chanSFD ! 2
WHILE counter <> 26666
counter := counter PLUS 1

-- EndOf: LCD_DisplayTime
References


References


- Etienne M. Gagnon, 1998, “SableCC, an Object-Oriented Compiler Framework” (Thesis), School of Computer Science, McGill University, Montreal


• JavaDoc, 1.5.0, Comparator, From: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html#sort(java.util.List,%20java.util.Comparator)


  http://citeseer.ist.psu.edu/page96hardwaresoftware.html


References


