Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognize the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.

Heejong Park
December 2015

Main Supervisor: Prof. Zoran Salcic
Co-Supervisors: Dr. Avinash Malik
Dr. Morteza Biglari-Abhari

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF
DOCTOR OF PHILOSOPHY IN ELECTRICAL AND ELECTRONICS ENGINEERING
Abstract

Most of today’s embedded systems are very complex. With decreasing hardware costs and rapidly developing semiconductor fabrication technology, designers are now able to build systems that incorporate a wide variety of computing technologies, the so called Cyber-Physical Systems (CPS). These systems, controlled by computer programs, continuously interact with their physical environments through a network of sensory input and output devices. Consequently, the operations of such embedded systems are highly reactive and concurrent. Since embedded systems are deployed in many safety-critical applications, where failures can lead to catastrophic events, an interdisciplinary approach such as mathematical logic and formal verification is employed in order to ensure correct behaviour of the control algorithm.

The size of these embedded applications is growing significantly each year. With increased time-to-market pressure, automatic and verified code generation from a design described in a high-level language is becoming of utmost importance. There has been significant effort in the development of a reactive programming language called SystemJ, which is based on the Globally Asynchronous Locally Synchronous (GALS) Model of Computation (MOC). With SystemJ, designers can describe a system using a number of asynchronously running concurrent processes called clock-domains. Each clock-domain is hierarchically composed of one or more reactions, which run concurrently but in lock-step with respect to the speed of the clock-domain. Reactions within a clock-domain communicate through broadcast based communication mechanism, while reactions in two different clock-domains communicate using a message passing mechanism. Similar to other reactive languages, SystemJ is designed based on formal mathematical semantics, allowing one to reason about the designed system before deployment. However, there have been very few studies on verification of programs developed in SystemJ and its GALS language counterparts, which makes it difficult to persuade designers to use such languages in practice. In this thesis, we propose an approach to close this gap by introducing a safety-critical subset of SystemJ called Safety-Critical SystemJ (SC-SystemJ) so that designers are able to automatically verify functional and real-time properties of the GALS system they develop with the language.
The original SystemJ language has been refined in order to make it amenable to both functional and real-time property verification. In particular, the SC-SystemJ semantics is based on big-step, which enables translating of the program into a network of finite state machines (FSM). The program can then be verified using many available and well-established techniques, which are based on automata theories. With the proposed approach, designers can verify functional correctness of the system on the generated FSMs, which is also an input for back-end code generation, thereby facilitating the What You Prove Is What You Execute (WYPIWYE) paradigm. In addition, an SMT based real-time analysis of the system has been introduced, which verifies whether the system, consisting of multiple communicating clock-domains, will generate output event(s) in response to input event(s) within certain time bounds as specified by the designer. Finally, a novel real-time programming technique has been introduced to the SC-SystemJ language via exact and non-exact real-time waits, which allows designers to describe real-time delays in the program behaviour like in traditional programming languages without need of external timers. Lastly, in order to support the new semantics as well as functional and real-time property verification, a tool-chain for the verification of the SC-SystemJ language has been developed, which includes the compiler, enabling correct-by-construction (WYPIWYE) design process.
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The technique/use of SystemJ Runtime Libraries presented in this paper are used in Chapter 1/Section 2.1

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Implementation, review and presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>50</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Writing and critical review</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
<tr>
<td>Panith Ratnayaka</td>
<td>A marginal co-author</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kevin I-Kai Wang</td>
<td></td>
<td>8/June/2015</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td></td>
<td>19/06/15</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The technique presented in this paper are used in Chapter 3 and Chapter 4/Section 4.1

| Nature of contribution by PhD candidate | Analysis, implementation, experiment and writing |
| Extent of contribution by PhD candidate (%) | 75 |

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>Research idea, implementation and writing</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>[Signature]</td>
<td>8/5/2013</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>[Signature]</td>
<td>19/06/13</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The techniques presented in this paper are used in Chapter 4/Sections 4.2 and 4.3

Nature of contribution by PhD candidate: Analysis, implementation, experiment and writing

Extent of contribution by PhD candidate (%): 80

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>Research idea, critical review and comments</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Click here

Click here

Click here

Click here

Last updated: 25 March 2013
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The techniques/benchmark programs presented in this paper are used in Chapter 4/Section 4.3 and Chapter 5/Section 5.6

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Implementation, experiment, reviewing and writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>Writing, critical review and comments</td>
</tr>
<tr>
<td>Muhammad Nadeem</td>
<td>Writing and experiment</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>[Signature]</td>
<td>08/06/2015</td>
</tr>
<tr>
<td>Muhammad Nadeem</td>
<td>[Signature]</td>
<td>08-06-2015</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>[Signature]</td>
<td>18/06/15</td>
</tr>
<tr>
<td></td>
<td>Click here</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Click here</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Click here</td>
<td></td>
</tr>
</tbody>
</table>
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The techniques presented in this paper are used in Chapter 5

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Analysis, implementation, experiment and writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>Research idea, critical review and comments</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

**Name** | **Signature** | **Date**
---------|---------------|---------
Avinash Malik | [Signature] | 8/06/2015
Zoran Salcic | [Signature] | 19/06/15

Click here

Click here

Click here

Click here

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The techniques presented in this paper are used in Chapter 6

Nature of contribution by PhD candidate: Analysis, implementation, experiment and writing

Extent of contribution by PhD candidate (%): 80

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>Research idea, critical review and comments</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

Certification by Co-Authors

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avinash Malik</td>
<td>[Signature]</td>
<td>8/06/2015</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>[Signature]</td>
<td>19/06/2015</td>
</tr>
</tbody>
</table>

Click here

Click here

Click here

Click here

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The techniques presented in this paper are used in Chapter 6.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Analysis, implementation, experiment and writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhenmin Li</td>
<td>Worst-Case Reaction Time (WCRT) Analysis</td>
</tr>
<tr>
<td>Avinash Malik</td>
<td>Research idea, critical review and comments</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhenmin Li</td>
<td>Zhen Li</td>
<td>03/06/2015</td>
</tr>
<tr>
<td>Avinash Malik</td>
<td>Malik</td>
<td>08/02/2015</td>
</tr>
<tr>
<td>Zoran Salcic</td>
<td>Salcic</td>
<td>19/06/15</td>
</tr>
</tbody>
</table>

Click here

Click here

Click here

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


The technique/benchmark programs presented in this paper are used in Chapter 6/Section 6.5

Nature of contribution by PhD candidate: Analysis, implementation, experiment and writing

Extent of contribution by PhD candidate (%): 80

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Saliclc</td>
<td>Critical review and comments</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Critical review and comments</td>
</tr>
<tr>
<td>Udayanto Dwatmojo</td>
<td>Critical review, comments and comparisons of existing approaches</td>
</tr>
<tr>
<td>Wei-Tsun Sun</td>
<td>Critical review and comments and writing</td>
</tr>
<tr>
<td>Avinash Malik</td>
<td>Critical review and comments</td>
</tr>
</tbody>
</table>

Certification by Co-Authors

The undersigned hereby certify that:

❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
❖ in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Saliclc</td>
<td>[Signature]</td>
<td>19/06/15</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>[Signature]</td>
<td>07/June/2015</td>
</tr>
<tr>
<td>Udayanto Dwatmojo</td>
<td>[Signature]</td>
<td>16/June/2015</td>
</tr>
<tr>
<td>Wei-Tsun Sun</td>
<td>[Signature]</td>
<td>07/June/2015</td>
</tr>
<tr>
<td>Avinash Malik</td>
<td>[Signature]</td>
<td>08/06/2015</td>
</tr>
</tbody>
</table>

Click here

Last updated: 25 March 2013
The completion of this work could not have been accomplished without the support of many people. First and foremost, I gratefully acknowledge my supervisor Prof. Zoran Salcic for his guidance and encouragement throughout my post-graduate studies. His constructive critiques and recommendations helped me to approach many problems encountered during the study with much broader perspective. Without his invaluable support it would not have been possible for me to complete this thesis.

I would also like to thank Dr. Avinash Malik for providing me helpful and wise advice throughout the research. He has always been a friendly mentor who spent countless hours with me discussing new ideas and emerging problems. I also appreciate Dr. Morteza Biglari-Abhari for his suggestions during many group meetings.

Last but not least, I would like to thank all of my family for their understanding and faithful support for many years. They have been pillars of strength to me during the hard and easy times.
Contents

Abstract i

Acknowledgements xi

Contents xiii

List of Figures xvii

List of Tables xxii

1 Introduction 1

1.1 Motivation 6

1.2 Research contributions 8

1.3 Thesis organisation 10

2 An Overview of the SystemJ Language 13

2.1 Globally asynchronous locally synchronous model of computation 13

2.2 The SystemJ kernel statements 17

2.2.1 SystemJ derived statements 19

2.3 An automated ice cream manufacturing facility case study 20

2.4 Asynchronous GRaph Code intermediate format 26

2.5 Causal cycles 29

3 Semantics of the SC-SystemJ Language 33

3.1 Motivation behind introducing SC-SystemJ 33

3.2 Syntax for the SC-SystemJ statements 35

3.3 Delayed signal semantics 37

3.4 Linear temporal logic 40

3.5 Transition semantics of SC-SystemJ programs 42

3.5.1 Transition semantics of synchronous clock-domain 43

3.5.2 Transition semantics of a network of clock-domains 50
## Contents

4 Compilation and Verification of SC-SystemJ Programs 57

4.1 Correct by construction code generation ........................................... 59

4.1.1 Generating a Mealy automaton for a synchronous clock-domain .......... 61

4.1.2 Generating a network of Mealy automata for GALS program .......... 71

4.1.3 Addressing specific issues .................................................. 73

4.2 Verification of SC-SystemJ programs .............................................. 75

4.2.1 Equivalence of descriptions .................................................. 78

4.2.2 Functional properties to be ensured for the ice cream manufacturing facility .................................................. 79

4.3 Experimental results for other benchmark programs .............................. 84

4.3.1 Benchmark descriptions ...................................................... 85

4.3.2 Verification results for the benchmark programs .............................. 86

4.3.3 Generated back-end code size, runtime, and compilation time comparisons .................................................. 88

5 Scheduling SC-SystemJ Programs for Guaranteed Response Times 91

5.1 Satisfiability Modulo Theories .................................................... 94

5.2 Motivating example ............................................................. 96

5.2.1 Reaction time ............................................................ 100

5.2.2 Response time ............................................................ 100

5.3 Specifying real-time bounds ..................................................... 102

5.4 The tool-chain flow for guaranteed response times ............................... 103

5.4.1 Compiling the motivating example into a network of FSMs .................. 105

5.4.2 Formalising programmer specified real-time constraints ....................... 106

5.4.3 Reaction time of a clock-domain in a GALS program ........................ 107

5.4.4 SMT formulation for a single processor system – scheduling a single FSM .................................................. 108

5.4.5 SMT formulation for a single processor system – scheduling multiple FSMs .................................................. 109

5.4.6 Extension to the multi-processor case ....................................... 114

5.5 Complexity analysis ............................................................ 115

5.6 Benchmark results .............................................................. 116

6 Exact and Non-exact Real-time Waits in a GALS Language 121

6.1 Motivation via examples of real-time systems programmed in SC-SystemJ 122

6.1.1 Programming using the non-exact real-time control construct: the wait_inbetween statement .................................................. 122

6.1.2 Programming using the non-exact real-time control construct: the wait_atleast statement .................................................. 124
6.1.3 Programming periodic tasks using the exact real-time construct: the \texttt{wait\_exact} statement ............................................... 125
6.2 Worst case reaction time and best case reaction time for mapping logical time to physical time .................................................. 126
6.3 Introducing Real-Time in SC-SystemJ ........................................ 128
  6.3.1 Semantics of the \texttt{wait\_inbetween} statement .................. 128
  6.3.2 Rewriting the \texttt{wait\_inbetween} statement .................... 129
  6.3.3 Finding the number of logical ticks \texttt{d} .......................... 129
  6.3.4 Finding the number of logical ticks \texttt{d} for the variants of \texttt{wait\_inbetween} state-
ment ..................................................................................... 131
  6.3.5 The tool-chain flow ......................................................... 133
6.4 Proofs of correctness, completeness and soundness ..................... 134
6.5 Experimental results .......................................................... 136
  6.5.1 Improving results .......................................................... 138
6.6 Discussion ............................................................................. 140
  6.6.1 Programming a periodic reaction using the \texttt{wait\_exact} construct .. 140
  6.6.2 Need for time analysable platforms ................................... 141
6.7 Other approaches to implementing real-time constructs in the reactive lan-
guages .................................................................................. 143

7 Conclusions and Future Work ................................................... 145
  7.1 Main contributions ........................................................... 146
  7.2 Possible future research directions ....................................... 148

A SC-SystemJ Grammar .............................................................. 151

B List of Publications .................................................................. 153
  B.1 Published Papers .............................................................. 153

References ................................................................................. 155
List of Figures

1.1 Overview of SC-SystemJ compilation flow ........................................... 9

2.1 A graphical overview of the SystemJ model ........................................... 14

2.2 Execution trace of clock-domain ...................................................... 15

2.3 Execution of reactions during a clock-domain tick ................................. 15

2.4 An effect of immediate and non-immediate qualifiers in the abort statement 18

2.5 An effect of weak qualifier in the abort statement .................................. 19

2.6 An automated ice cream factory distributed controller ............................. 21

2.7 Graphical overview of the ice cream manufacturing facility implemented in SystemJ ................................................................. 21

2.8 The automated ice cream control logic in SystemJ ................................. 24

2.9 AGRC graph for the SSC clock-domain in Figure 2.8 ............................. 28

3.1 Different execution traces for instantaneous and delayed signal semantics . 38

3.2 Two infinite sequences of states that hold \( X_p \) and \( p_1 U p_2 \) ................ 41

3.3 Two infinite sequences of states that hold \( F_p \) and \( G_p \) ..................... 42

3.4 Inductive definitions for SC-SystemJ statements .................................... 45

3.5 Inductive definitions for SC-SystemJ statements continued ....................... 46

3.6 Definitions for SC-SystemJ data statements ........................................ 48

3.7 Reactivity of the SC-SystemJ transition systems .................................... 49

3.8 Determinism of the SC-SystemJ transition systems ................................ 50

3.9 Sender and Receiver communicating via ACK and REQ signals .............. 51

3.10 Three possible scenarios that might occur during rendezvous ................ 52

3.11 Deadlock due to preemption ........................................................... 53

3.12 Non-deterministic GALS program .................................................... 55

3.13 SC-SystemJ GALS program generating different output signals for an input signal \( A \) ................................................................. 55

4.1 Overall WYPIWYE SC-SystemJ program compilation approach for functional verification .......................................................... 60
4.2 An example of runs for a Generalized Büchi Automaton .................. 62
4.3 Example transition system generated from the SC-SystemJ semantics ... 63
4.4 Derivation tree deducing $\texttt{Inst}$ for SC-SystemJ program in Figure 4.3a ... 64
4.5 Derivation tree deducing $\texttt{Enter}$ for SC-SystemJ program in Figure 4.3a ... 64
4.6 Derivation tree deducing $\texttt{Term}$ for SC-SystemJ program in Figure 4.3a ... 64
4.7 Derivation tree deducing $\texttt{Move}$ for SC-SystemJ program in Figure 4.3a ... 65
4.8 Labelled Generalized Büchi automaton generated from the SC-SystemJ program in Figure 4.3a ... 65
4.9 Creating a deterministic Mealy automaton ........................................ 66
4.10 Generated automaton and C code for SC-SystemJ example in Figure 4.3a ... 71
4.11 Generating C code for SC-SystemJ example with a network of clock-domains ... 72
4.12 Generated automaton Promela code for SC-SystemJ example in Figure 4.3a ... 76
4.13 Generated Promela code for SC-SystemJ GALS program in Figure 4.11a ... 77
4.14 A closed loop manufacturing controller system for verifying liveness properties ... 82
4.15 Verifying functional property of the ice cream manufacturing facility using $\texttt{iSPIN}$ .......................................................... 83
4.16 The heart and the pacemaker ............................................................. 86
4.17 Comparison between the AGRC-based and the automata-based compiler in terms of generated code size and average reaction times ............. 88
4.18 Compilation time comparison between the AGRC-based and the automata-based compiler ................................................................. 90
5.1 Mixed job scheduling problem ......................................................... 95
5.2 Execution trace of jobs satisfying end time constraints ....................... 96
5.3 An automated ice cream manufacturing facility simplified ..................... 97
5.4 Updated SC-SystemJ implementation of ICMF for response time analysis ... 98
5.5 SC-SystemJ control logic controlling ICMF ......................................... 98
5.6 An execution trace of CD1 ............................................................... 99
5.7 Response times for A to D and A to T .............................................. 101
5.8 An example of response time constraints .......................................... 102
5.9 The extended overall design flow to guarantee response times ............. 104
5.10 FSM representation of the ICMF in Figure 5.4 .................................. 105
5.11 Adding transition times to the edges ................................................. 107
5.12 Four-phase handshake and its encoding in SMT ................................ 111
5.13 State representation of parallel rendezvous communications ............. 112
5.14 A schedule for ICMF benchmark with response times. All clock-domains are allocated to a single processor .................................. 113
5.15 A schedule for ICMF benchmark with each of the three clock-domains running on their individual processors ........................................ 114
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.16</td>
<td>Z3 SMT solver runtime with varying # of constraint clauses</td>
<td>117</td>
</tr>
<tr>
<td>5.17</td>
<td>Extending benchmark example for investigating scalability</td>
<td>118</td>
</tr>
<tr>
<td>6.1</td>
<td>Programming a human responsiveness system (HRTCS) with <code>wait_inbetween</code></td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>Programming a timeout with <code>wait_atleast</code></td>
<td>124</td>
</tr>
<tr>
<td>6.3</td>
<td>Programming a periodic task with <code>wait_exact</code></td>
<td>125</td>
</tr>
<tr>
<td>6.4</td>
<td>Simple SC-SystemJ example and corresponding MoC</td>
<td>127</td>
</tr>
<tr>
<td>6.5</td>
<td>The rewrite of <code>wait_inbetween</code> construct</td>
<td>128</td>
</tr>
<tr>
<td>6.6</td>
<td>The pictorial representation of the solution for the HRTCS system</td>
<td>131</td>
</tr>
<tr>
<td>6.7</td>
<td>The tool-chain flow of SC-SystemJ compiler with waits</td>
<td>133</td>
</tr>
<tr>
<td>6.8</td>
<td>Replacing external timer with <code>wait_atleast</code> construct</td>
<td>137</td>
</tr>
<tr>
<td>6.9</td>
<td>Accuracy of the results</td>
<td>138</td>
</tr>
<tr>
<td>6.10</td>
<td>Improving WCRT by shortening a critical path</td>
<td>139</td>
</tr>
<tr>
<td>6.11</td>
<td>Reduction in upper bound relaxation with improved WCRT</td>
<td>139</td>
</tr>
<tr>
<td>6.12</td>
<td>Extra pause statements in periodic tasks</td>
<td>140</td>
</tr>
<tr>
<td>6.13</td>
<td>Interaction of preemption and external timers</td>
<td>141</td>
</tr>
<tr>
<td>6.14</td>
<td>Resolution of external timers</td>
<td>142</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The SystemJ kernel statements</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>The SystemJ derived statements</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>The semantics of the AGRC nodes</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>SC-SystemJ kernel statements in comparison with the SystemJ kernel statements</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>SC-SystemJ derived statements in comparison with the SystemJ derived statements. Expansion rules for all other SystemJ derived statements shown in Table 2.2 are identical in SC-SystemJ</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>A list of verified LTL properties for the automated ice cream facility</td>
<td>82</td>
</tr>
<tr>
<td>4.2</td>
<td>Results of the property verification</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>Performance of the compiler</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>An overview of the benchmark programs</td>
<td>116</td>
</tr>
<tr>
<td>5.2</td>
<td>Benchmark results</td>
<td>116</td>
</tr>
<tr>
<td>6.1</td>
<td>An overview of the benchmark programs</td>
<td>136</td>
</tr>
<tr>
<td>6.2</td>
<td>Actual delays obtained for wait constructs in the benchmark programs based on their BCRT and WCRT</td>
<td>137</td>
</tr>
</tbody>
</table>
The advancement of integrated semiconductor technologies over the past half-century has enabled engineers to develop a great number of applications that were quite difficult to even imagine in the past. Today’s integrated circuit (IC) designs, such as microprocessors and flash memory chips, incorporate billions of transistors per die capacities. Moreover, cost per transistor in micro-chips has been drastically reduced, leading to mass production of more advanced digital electronics. Along with this technological evolution, computing systems have become more pervasive in people’s daily lives where continuously capturing input events and generating responsive outputs are essential. In general, such types of systems are characterised by constrained resources, mainly due to restricted operational environments, and are categorised as embedded systems. They are computer systems that are application-specific and dedicated to performing a specialised tasks as part of a larger system.

Traditionally, embedded systems were programmed using low level languages, such as C or assembly, due to the restricted performance and memory constraints of the target hardware. Designers often had to access hardware devices directly or write fine-tuned code in order to exploit the limited resources. Hence, designing and programming embedded systems is largely different from desktop computing since heavyweight abstraction layers and virtualisation techniques are not yet a viable option [1]. Moreover, unlike desktop computing, most embedded software is tailored to a particular application domain, for example: avionics, industrial control, medical and automotive systems, etc. Therefore,
developers of embedded systems are required not only to have good programming skills but also broad understanding and knowledge of their field of work \[1\], which makes developing embedded software systems more challenging and usually requiring domain specific experts.

Embedded system designers often have to consider physical characteristics of the real-life environment. For example, when a security system, designed to trigger an alarm in case of intrusion, senses movement within an operating area it should wait for a certain time until an authorized person provides a valid PIN number. Such systems, apart from meeting their functional behaviour (e.g. the system will trigger an alarm), are also required to satisfy real-time constraints in order to meet system specification (e.g. the system will set-off an alarm after a finite time-out). Usually these constraints are dictated by the behaviour of the environment and missing a deadline may result not only in degradation of system performance but also catastrophic events such as loss of life or property. These types of systems are called \textit{real-time systems}. Normally, real-time systems are classified into two types \[2\]:

\begin{description}
\item[Soft real-time systems] are able to tolerate some occasional deadline misses that will only degrade the system performance without consequences on overall system functionality. These systems are unable to guarantee an absolute performance in terms of meeting real-time constraints. Examples of soft real-time systems are: games, digital TV broadcasting, and voice over internet protocol (VOIP), etc.
\item[Hard real-time systems] cannot tolerate any deadline misses. Even a single deadline miss can lead to system failure with catastrophic consequences. Therefore, they are both \textit{safety-critical} and \textit{hard real-time systems}. Examples are: pacemakers, air traffic control systems, and anti-lock brakes, etc.
\end{description}

In the domain of real-time embedded computing, especially for describing concurrent behaviours, there are two major approaches to specification and implementation: (1) Language based, like Ada \[3\], and (2) RTOS based with a sequential language. Ada is a general-purpose strongly-typed language which has been widely used in many defence and aerospace industries. It was designed with real-time systems in mind; the language has built-in support for task scheduling and synchronisation, real-time facilities (e.g. time-outs), and data sharing through message passing. In addition, the Ravenscar profile \[4\], which is introduced in Ada 2005, further restricts the tasking model of Ada guiding programmers to develop applications amenable to high-integrity certification standards such as DO-178B/ED-12B \[5\]. For example, a software for En Route Automation Modernization (ERAM) Air Traffic Control (ATC) system \[6\]. This software replaces almost 40-year old air traffic control centres used in the United States, and it is written mostly in ADA.
Use of a real-time operating system (RTOS) is another well-used solution in developing real-time applications. There exist both commercial and open-source RTOS; notably µCOS-II/III [7] and FreeRTOS [8]. These RTOS have very small memory footprints (typically between 5 to 20 Kilobytes), and support various real-time features including prioritised real-time task scheduling, and minimal latencies for the interrupt service routine (ISR) and task switching, making them amenable to many safety-critical applications. Nevertheless, an application that uses a RTOS introduces other potential issues such as priority inversions and deadlocks, race conditions etc., which complicates verifying correctness of the design and implementation [9].

With growing complexity of design and increasing time-to-market pressures, there have also been efforts to develop new programming languages based on novel reactive paradigms, such as Esterel [10], Signal [11], and Lustre [12]. These so called synchronous languages are designed for development of complex reactive systems. Reactive systems [13] are computer systems that continuously capture input events, update their internal state, and generate outputs at a speed determined by the environment. Unlike general purpose programming languages, the computation model of the reactive languages is based on rigorous mathematical foundations and hence, applications written in these languages can be formally verified for functional correctness. Esterel is specifically designed for describing control-flow and concurrent computations using reactive statements, which naturally abstracts away conventional concurrent programming techniques such as thread synchronisation, preemption and data sharing. However, as Esterel has its root in hardware description languages (HDLs), it lacks solid foundations for describing and managing complex data-structures. Signal and Lustre, on the other hand, are synchronous data-flow languages more suited for describing the data-driven computations that model programs as a network of connected data-processing nodes. On the other hand there also exist an approach, such as in [14], which extends Java with synchronous paradigm through libraries.

Reactive systems are most often considered to be real-time in the sense that systems “react” to requests from an outside world in a consistent and timely manner. Indeed, depending on the type of applications (soft or hard real-time), failure to respond to input events can lead to unwanted consequences. Therefore, it is desirable that the system designs are verified against their specifications through formal mathematical techniques [15] in order to prevent deployed systems from misbehaving. One of the verification challenges for reactive systems, for instance, has been to verify the system’s ability to serve requests from environment under any circumstances. There have been many recent research proposals for verifying timing properties of programs developed with pure synchronous approaches [16, 17, 18, 19]. All these works focus on guaranteeing that reactive systems do not miss any input from environment. However, most of these works overlook the
importance of analysing the response time of a system; the timing guarantees between 
the arrival of input events and generation of the corresponding output events.

Although, languages such as Esterel, Signal and Lustre have their merits, the pure 
 synchronous approach does not always fit well with designing today’s mostly distributed 
and largely heterogeneous systems. For example, designers might want to decouple dif-
f erent parts in a system in order to save power, e.g. not all components in a system 
are required to run at the system’s fastest clock speed, or to remove synchronisation 
issues due to a large design space, e.g. clock skew. To overcome these issues, a new 
paradigm, called Globally Asynchronous Locally Synchronous (GALS), was proposed first 
in [20]. The design principle of a GALS system is a combination of synchronous and 
asynchronous approaches, where individual concurrent parts of a system can be either 
tightly connected on a local level, following the synchronous paradigm, or more loosely 
coupled at the top level, running asynchronously to each other.

Early works on GALS systems were mainly in the hardware design domain such as 
those described in [20], which addressed metastability of hardware blocks driven by dif-
f erent clocks by using pausable clocks. The main idea of the GALS model in the hardware 
domain is building a system that consists of hardware blocks running with their own clock, 
connected with each other through unidirectional point-to-point or global abstracted in-
terconnect mechanisms.

On the other hand, there also exist many programming languages, which can capture 
features of GALS systems directly at the language level. For example, SystemJ [21] is 
a language based on the GALS Model of Computation (MoC), specialised in describing 
both control and data-driven operations in a system. Communicating Reactive Processes 
(CRP) [22], proposed by Berry et al., is also a language for developing GALS systems 
but provides minimal data support similar to Esterel. Communicating Reactive State 
Machines (CRSM) proposed by Ramesh [23] translates networks of finite state machines 
into reactive processes whose semantics are based on the boolean automata model of Ar-
gos [24]. Each CRSM process is driven by its own clock, and has an independent interface 
to the environment; this makes a system modelled in CRSM follow the GALS MoC. More 
recently, a C based library called LibGALS [25] has been introduced which extends C’s 
sequential programming model with the GALS MoC and reactivity. All aforementioned 
GALS approaches provide a means of communication between asynchronous processes 
through rendezvous style message passing [26] which guarantees message delivery be-
tween a sender and a receiver. Lastly, authors in [27] defined a constructive operational 
semantics which combines Quartz [28] and Signal. Their framework allows one to model 
ill-specified program behaviours, such as causality, as a formal verification problems.

Both pure synchronous and GALS languages are designed to target reactive and real-
time systems. Nevertheless, they still lack in many real-time features found in RTOS-
based alternatives. For instance, many applications written in any of the aforementioned GALS languages that require measuring real-time intervals are heavily dependent on an external device outside of the designed system, such as physical clocks. It is mainly because their MoC is based on the logical instant of time (in other words a tick) and expressing the notion of physical time is not directly supported in the language syntax or semantics. Furthermore, lack of tools for verifying functional correctness of GALS systems puts additional burden on designers for ensuring their system will not fail.

The primary focus of this thesis is on introducing real-time features and verification methods for programs designed in the GALS MoC, in particular, SystemJ, which has been chosen as the starting point for this research due to several reasons:

1. Unlike CRP or other similar GALS variants, whose primary targets are control-oriented applications, SystemJ is designed to model both control and data-driven aspects of a system. In addition, the message passing mechanism between asynchronous processes is much simpler and more efficient in SystemJ compared to CRP or LibGALS which require additional processes. Moreover, as we will see later, CRP’s rendezvous mechanism is ill-suited for verification.

2. SystemJ is a language-based approach in which the compiler automatically compiles the program and generates executable code that follows GALS MoC. If the compiler detects that a program does not comply with semantics, it will reject the program and possibly suggest alternative approaches. Furthermore, introducing new features and verification approaches can be naturally incorporated in the compiler. On the other hand, library-based approaches like LibGALS are more error-prone in this regard as programmers have more freedom but also greater responsibility regarding how they write a program; even if it does not adhere to the GALS MoC, as long as the host language (in this case C) may still allow it. Moreover, the program behaviour described in the host language is very difficult to verify with the library-based approach.

3. The language has been used in many large applications [29, 30, 31, 32], demonstrating its capability for designing real world systems.

4. The language has been actively maintained by many researchers and continuously improved over several years [33].

In the following sections, the research motivations and contributions are presented.
1.1 Motivation

Development of real-time embedded applications has always been a challenge, since meeting timing requirements adds complexity to the development process. Moreover, verifying correctness of such applications becomes even more difficult in the presence of concurrency where all the classical problems found in concurrent programming, such as deadlocks and livelocks, may lurk, hidden, in a program. In order to address such issues, higher-level design languages, based on formal mathematical foundations, have emerged. Such languages allow formal reasoning about the behaviour of the system, and typically allow programmers to develop a whole system in a single design paradigm, and thus are often called system-level design languages. For example, Globally Asynchronous Locally Synchronous (GALS) languages such as SystemJ have been one of the notable works in development of a large concurrent systems. However, most of the current GALS languages, including SystemJ, lack in both functional and real-time verification techniques and tools as well as facilities for describing physical time as first class constructs in the language.

The goal of this research is to introduce a way to verify functional and timing properties of a system, developed in a subset of the SystemJ language, called Safety-Critical SystemJ (SC-SystemJ), targeted towards hard real-time, safety-critical applications. Motivations of the research can be further refined as follows:

1. Need for a correct-by-construction approach that is scalable for GALS reactive systems

Programs developed in the SystemJ language have been claimed to be formally verifiable in many articles [21, 31, 34], because it is based on a formal semantics. However, claims have not been substantiated because of lack of tools to verify functional properties of the designed system.

Verification of safety-critical systems differs from standard testing approaches, such as unit-testing, since verification requires complete state exploration in order to guarantee that a property of the developed system holds under all possible inputs to the system. Normally, verification is carried out on the model of the developed system. However, if the implementation differs from the actual model then the verification is of little value. Hence, a compilation approach adhering to the What You Prove is What You Execute (WYPIWYE) [35] paradigm needs to be developed.

One of the main reasons for using formal languages in designing embedded systems is the ability to guarantee functional correctness of the designed/implemented system. Thus, it is essential that there should be verification tool-chains available for SC-SystemJ programmers.
2. Need for a novel approach for guaranteeing response times in GALS reactive systems

In the domain of pure synchronous programs there have been many attempts to guarantee that the program meets all its deadlines. For example, in [36], Esterel is extended with timed annotations used in TimeC [37], which enables specifying timing constraints in Esterel programs. In this approach, a block of Esterel code is wrapped around with ghost signals specifying beginning and end locations of code to be analysed. A program is then compiled into an automaton, representing all the states and possible transitions of the Esterel program; this is used to validate the timing constraints (i.e. annotations) on a given execution architecture. However, the proposed verification method cannot handle multiple state transitions i.e. the ghost signals cannot enclose Esterel statements which consume logical time. Moreover, adding ghost signals will inevitably increase compiled code size, and thus degrade program performance.

There are also integer linear programming (ILP) based approaches [16] for guaranteeing real-time properties of synchronous programs. In this work, a synchronous program is formulated as an extended ILP problem called ILPc, which addresses the state explosion problem in traditional ILP based approaches [38, 39] due to concurrent processes in Esterel programs. The main goal of this work was to find how fast the Esterel program can capture input events arriving from the environment, hence guaranteeing the Worst Case Reaction Time (WCRT) of the program. Nonetheless, this work primarily focuses on synchronous programs which cannot be directly applied to GALS approaches due to the asynchronous interleaving semantics. Lastly, similar to the work in [36], this work does not address the case when response to an input event across multiple instances of logical ticks needs to be bounded.

Timing analysis of GALS programs is different from synchronous approaches; one needs to incorporate asynchronous concurrency and the message passing communication mechanism in the analysis process. In particular, this thesis focuses on response time analysis of GALS programs. Conceptually, the response time analysis is to find a timing relationship between input and output events, which may involve execution time of a program across multiple logical ticks and multiple asynchronous processes. Therefore, unlike traditional approaches targeting only synchronous programs, the response time analysis is not limited to a single logical tick. Lastly, response time analysis incorporates asynchronous features in the GALS paradigm, which are not found in pure synchronous languages.

3. Need for explicit real-time control in the GALS language

One of the prominent features in real-time computing is the ability to associate program behaviours with physical time through facilities such as timeouts and pe-
iodic activities. Most of these features are used to control the execution speed of a program in order to align its operation with certain behaviours of the physical environment. This is essential in developing real-time applications because it is common that the speed of the physical environment is much slower than a software program running on digital hardware. For example, consider a program which controls the motion of a robot by applying different pulse-width modulation (PWM) duty-cycles; the time duration of the PWM signal needs to be precisely controlled in order to speed up or slow down the movement of the robot.

Normally, physical time is incorporated in GALS or synchronous programs with the help of external timers; the program emits a signal to a timer, which also usually carries the timing information, and waits until the timer expires and signals back to the program. Although, this method works in most languages, needing additional resources for a timer is not always an appealing solution. Moreover, it will be shown later why external timers do not interact well with synchronous and GALS programs – in fact, not all timing behaviours can be guaranteed using this approach.

1.2 Research contributions

The main contribution of this thesis is the introduction of a safety-critical subset of SystemJ that addresses unexplored opportunities in verification and real-time domains. In particular, this subset is called Safety-Critical SystemJ (SC-SystemJ). An overview of SC-SystemJ compilation flow is shown in Figure 1.1 in the comparison with the original SystemJ compilation flow. Originally [21], the SystemJ program was converted into an intermediate representation which is based on Micro-Step Structural Operational Semantics (SOS) [40]. Micro-step semantics provided a way to compile SystemJ programs into an efficient executable while preserving the GALS MoC of the designed system.

However, micro-step-based semantics are too fine-grained, i.e. the logical tick transitions required for verification are not explicit. Moreover, the data transformations and related state changes are closely intertwined with control-flow, thus making verification challenging. The new approach is based on automata semantics [41], which represents a whole SC-SystemJ program as a finite state machine. As shown in Figure 1.1, real-time and functional verification of the SC-SystemJ program will be based on this new intermediate format. The primary contributions of this thesis are as follows:

1. A new semantics for SC-SystemJ that is amenable to functional and real-time property verification

The current compilation approach of the SystemJ language converts a SystemJ pro-
1.2 Research contributions

A new approach for verification of functional properties of the SC-SystemJ program built upon the new formal semantics is introduced. The new semantics enable compilation of a SC-SystemJ program into finite state automaton. This automaton is then verified against a set of properties specified in temporal logic \[43\] using a model checker \[44\], which explores the complete state space of the program. This approach incorporates verification of programs underpinned with both synchronous and GALS MoC.

3. A novel approach for timing analysis of the SC-SystemJ program that makes it suitable for designing real-time embedded systems

This thesis proposes formalisation of real-time analysis in the GALS setting in which...
the program consists of multiple asynchronous and synchronous behaviours and generates a delayed response to input events. The approach will be demonstrated using SC-SystemJ programs which guarantee response times of the system. Unlike traditional WCRT analysis [45], which mainly focuses on how fast a program can capture the input events, the proposed approach emphasises timing requirements between input and output event relationships, which are crucial particularly for real-time control systems. It is also shown that the proposed approach is scalable for multi-processor distributed memory systems.

4. **Expressing real-time as first class language constructs in SC-SystemJ programs**

In the original semantics of the SystemJ language, time is only logical; one can not alter control-flow of the SystemJ program using the notion of physical time (i.e. seconds, milliseconds, etc.). Due to the lack of this feature, it has been difficult for SystemJ programmers to develop applications that require explicit real-time constraints imposed by the environment (e.g., driving a motor for a certain period of time). This thesis introduces exact and non-exact real-time waits in the SC-SystemJ language. These additions allow system designers to explicitly use, at the specification level, not only logical time but also real-time in order to control program execution.

5. **Efficient compilation and automated generation of the verified executable code**

The SC-SystemJ program can be verified and compiled to various target execution platforms. Since the compilation strategy for the automata-based semantics is largely different from the original micro-step-based semantics, the tool-chain diverges from the earlier compilation approach. The generated code is more efficient in terms of execution time and size than the original micro-step-based approach.

### 1.3 Thesis organisation

[Chapter 2] gives an overview of the original SystemJ language including its Model of Computation and syntax. In the same chapter, a case study called automated ice cream manufacturing facility is introduced; this demonstrates how SystemJ can be used for developing reactive embedded systems. The chapter then briefly illustrates how the SystemJ program is transformed into the original internal representation called Asynchronous GRaph Code (AGRC) format, which defines the program behaviour as a sequence of micro-steps, and also discusses several semantic issues in the original SystemJ MoC’s synchronous communication model.
Chapter 3 presents the semantics of the SC-SystemJ language based on *big-step* transitions of program states. In addition, a new communication model is introduced in order to address the issues explained in Chapter 2. Lastly, it is shown how the SC-SystemJ program is converted into a network of finite state machine (FSM), which is a new intermediate format used for back-end code generation.

Chapter 4 introduces the new compilation and verification approach for SC-SystemJ programs. In particular, generation of C and Promela [44] code for both pure synchronous and GALS SC-SystemJ programs is shown. Promela code is used for verifying the SC-SystemJ program via the SPIN model checker [44] while the C code, which can be compiled and deployed on a target execution platform, is generated from Promela once the program is proven correct, thereby adhering to the WYPIWYE paradigm.

Following that, Chapter 5 gives the novel method for response time analysis of SC-SystemJ programs using a Satisfiability Modulo Theories (SMT) [46] based scheduling approach.

Chapter 6 introduces new real-time *wait* constructs in the SC-SystemJ language, which provides exact and non-exact real-time control mechanisms. This thesis finally concludes in Chapter 7 with discussion of some possible future research ideas.
An Overview of the SystemJ Language

This chapter will introduce the system-level design language SystemJ. Section 2.1 presents the model of computation of SystemJ. Section 2.2 introduces the SystemJ kernel and derived statements, and describes their informal semantics. Section 2.3 introduces a SystemJ case study called automated ice cream manufacturing facility, which is a running example used throughout this thesis. Section 2.4 describes an intermediate format used in the original micro-step-based compilation approach. Finally, the chapter concludes in Section 2.5 with a discussion of the causality issues in the original semantics of the SystemJ language.

2.1 Globally asynchronous locally synchronous model of computation

SystemJ [21, 47, 48] is a system-level language designed for modelling and implementing reactive and concurrent software systems that always follow the Globally Asynchronous Locally Synchronous model of computation (GALS MoC). A GALS model provides a means to construct a system that consists of loosely coupled components on a global level which run asynchronously to each other. Furthermore, closely related parts of a system, which communicate with each other more frequently, can be grouped together on a local level within each asynchronous component.

Figure 2.1 shows a graphical illustration of a complete SystemJ program. The language
provides two concurrent entities which can be used for describing global and local levels of a system:

1. **Clock-domains**: A SystemJ program consists of one or more asynchronous-concurrent behaviours (processes), called clock-domains (CD). For example, the system shown in Figure 2.1 models a small sensing and controlling system where two clock-domains The **Sensor** and **Controller** are executed asynchronously to each other. The **Sensor** clock-domain gathers information from the external environment, processes the information, and transfers the results to the **Controller** clock-domain. The **Controller** clock-domain then generates appropriate control signals which are emitted back to the external environment. Clock-domains, which represent the top-level of a SystemJ program, execute asynchronously at their own pace, defined by their own notion of logical ticks (or just ticks). Ticks refer to a sequence of discrete instants that capture the logical time of the SystemJ clock-domain. Figure 2.2 illustrates execution of a SystemJ clock-domain based on logical ticks (i.e. $T_1, T_2, \ldots, T_5$). At each tick, clock-domains initiate communication with the external environment in which they take in input events and, based on these inputs, produce corresponding output events, as shown in Figure 2.2. From the environment’s point of view, a tick is considered instantaneous, i.e., all the computations are executed in zero time. This execution model, for individual clock-domains, is based on the perfect synchrony hypothesis [10] which is adopted from Esterel. The perfect synchrony hypothesis assumes that the speed of the clock-domain is infinitely faster than its respective environment such that the environment cannot detect any delay caused by the execution of actual program logic inside the clock-domain.

2. **Reactions**: Each clock-domain can be refined into one or more synchronous and
concurrent sub-behaviours, which are called reactions. For example, in Figure 2.1 the synchronous parallel operator || (e.g. R1||R2) is used to build clock-domains from multiple reactions. Reactions can be composed hierarchically, by forking child reactions from the parent reactions, effectively implementing behavioural hierarchy. Reactions within a single clock-domain communicate via a synchronous broadcast mechanism using signals, whereas reactions in two different clock-domains communicate through a rendezvous-style message passing mechanism using channels. Multiple reactions in the same clock-domain execute concurrently and advance their states in lock-step, following the logical clock (tick) of the clock-domain they belong to. Therefore, every reaction inside a clock-domain needs to finish its execution in order for the whole clock-domain to progress to the next tick. Consider Figure 2.3 which shows execution of three reactions during a clock-domain tick $T_1$. Assume all three reactions start their execution at the beginning of $T_1$. However, the time to complete each reaction varies; hence, the execution time of the reactions are all different. In Figure 2.3 Reaction-2 finishes its execution first followed by Reaction-1. However, they cannot advance to the next tick since Reaction-3 has not yet finished its execution, and is hence not yet synchronized with the two other reactions.

SystemJ provides two abstract communication objects, which abstract the underlying physical devices and communication protocols from the system designer:

1. *Signals*: A signal is represented by its binary status, which can be either true
An Overview of the SystemJ Language

(present) or false (absent). The reaction that emits a signal sets its status to true for only the current logical clock tick until the beginning of the next tick. The emitted signal is instantaneously broadcasted within the clock-domain and is visible to all reactions within the clock-domain. The status is reset to false at the beginning of the next logical clock tick of the clock-domain. The first use of a signal is to facilitate communication between reactions within a single clock-domain. Referring back to Figure 2.1, the signal S1 is emitted by reaction R4 and can be captured, instantaneously, by both reactions R3 and R5 (i.e. S1 is broadcasted within the clock-domain). In addition to the binary status, a signal may also carry a value that can be of any Java object or primitive type; such a signal is called a valued signal. If a signal is a valued signal, the status along with its value are emitted together. In addition, signals in SystemJ are also used as the communication mechanism between reactions and the external environment (e.g., I1, I2 and O1, O2 in Figure 2.1); these signals are called interface signals. Interface signals have a direction, i.e. input or output, and are used to exchange information with the physical environments. For example, a timeout from a timer, sensor and actuator IOs from serial buses, and communication events such as incoming socket messages, etc. Inputs for all reactions within a clock-domain are sampled at the beginning of the logical clock tick and the corresponding outputs are generated at the end of the tick.

2. Channels: SystemJ clock-domains are asynchronous with respect to each other, hence, signal broadcast with its limited signal life-time is not a reliable communication mechanism between reactions from different clock-domains. In order to prevent unreliable communication another communication object, called a channel, is used to exchange data between reactions in different clock-domains. Channels implement a rendezvous-style message-passing mechanism adopted from CSP [26] (e.g. channels CH1 and CH2 in Figure 2.1) which ensures the delivery of the message. Channel communication is uni-directional and point-to-point. Reactions from both sending and receiving clock-domains should be ready in order to commence and complete the message transfer i.e. it is a blocking operation. Therefore, sending and receiving reactions will not proceed further until the rendezvous and data transfer is accomplished.

Platform dependent libraries, such as initialisation code and underlying communication methods for SystemJ channels and signal handling are provided and maintained separately as the SystemJ Run-Time Support (RTS) library [30] [49], which bridges the physical world to the signals and channels in SystemJ. For instance, referring to Figure 2.1, The SystemJ RTS samples the physical signals (devices) P1 and P2 at the beginning of the tick, and then forwards them to signals I1 and I2, respectively. Similarly, P3 and P4
2.2 The SystemJ kernel statements

A complete list of SystemJ kernel statements is shown in Table 2.1. Sequential SystemJ statements are separated by a semicolon ";" similar to C and Java. The tick boundaries are explicitly specified with pause – every reaction in a clock-domain has to reach one of the pause statements in order to begin the next tick. It should be noted that only pause or any other statements enclosing pause consume logical time in SystemJ. When a clock-domain pauses, all the output signals emitted in the current tick become available to the environment, and all the input signals to be processed in the next tick are captured from the environment. In addition, signals allow communication between reactions within the same clock-domain and their statuses can be set to true for one tick using the emit statement. The while statement is a temporal loop that must contain at least one pause inside its body. Software exceptions are done through trap and exit; this preempts a program whenever it hits the exit statement inside a trap block p. SystemJ reactions are composed using the synchronous parallel operator (||) and make transitions in lock-steps with the clock-domain’s tick.

There are also statements that allow programmers to change control-flow of a program based on presence or absence of a SystemJ signal. For example, the present-else statement is similar to if-else in traditional programming languages, but it branches program execution depending on the status of a signal or the evaluation of signal expressions. The abort(S) p ; q statement preempts the on-going execution of a program body p if the signal expression S is evaluated to true. As a result, a program enters q devices are actuated whenever signals O1 and O2 are emitted at the end of the tick.

<table>
<thead>
<tr>
<th>Type</th>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>p1;p2</td>
<td>Statements p1 and p2 in sequence</td>
</tr>
<tr>
<td></td>
<td>pause</td>
<td>Consumes a logical instant of time (a tick boundary)</td>
</tr>
<tr>
<td></td>
<td>[input/output] [type] signal S</td>
<td>Declaring a pure or valued signal</td>
</tr>
<tr>
<td></td>
<td>emit S[(exp)]</td>
<td>Emitting a signal with a possible value</td>
</tr>
<tr>
<td></td>
<td>while(true) p</td>
<td>Temporal loop</td>
</tr>
<tr>
<td></td>
<td>present(S) p1 else p2</td>
<td>If signal S is present do p1 else do p2</td>
</tr>
<tr>
<td></td>
<td>abort ([immediate] S) p</td>
<td>Preempt p if S is present</td>
</tr>
<tr>
<td></td>
<td>suspend ([immediate] S) p</td>
<td>Suspend for 1 tick if S is present</td>
</tr>
<tr>
<td></td>
<td>trap (T) p [do q]</td>
<td>Software exception preempts p</td>
</tr>
<tr>
<td></td>
<td>exit (T)</td>
<td>Throw a software exception</td>
</tr>
<tr>
<td></td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td>[input/output] [type] channel C</td>
<td>Declaring input or output channel</td>
</tr>
<tr>
<td></td>
<td>send C[(exp)]</td>
<td>Sending data over the channel</td>
</tr>
<tr>
<td></td>
<td>receive C[(exp)]</td>
<td>Receiving data over the channel</td>
</tr>
<tr>
<td>Other</td>
<td>#C</td>
<td>Retrieving data from a valued signal or channel</td>
</tr>
<tr>
<td></td>
<td>jterm(p)</td>
<td>Java data-oriented computation</td>
</tr>
</tbody>
</table>

credible
and continues execution. Whenever the signal expression is evaluated to true, suspend preempts the program body, which it is enclosing, for one logical tick. But, unlike abort, suspend does not completely terminate the enclosing program body, and the program continues execution in the following tick from where it was last paused.

The abort and suspend statements can optionally have the immediate qualifier. They give programmers more precise control of program execution. When the statements are written with immediate, they check for signal presence from the very first tick i.e. when program first executes the statement. Otherwise, the check for the signal is delayed by one tick. For example, Figure 2.4 shows two simple programs whose output sequences are different due to the use of the immediate qualifier. Both examples emit the signal $S$ first, which is captured by the abort construct in the following statement. In Figure 2.4a, abort($S$) does not check the signal emission since the statement is non-immediate. As a result, the program also emits the signal $O$ in the same tick. On the other hand, in Figure 2.4b, abort(immediate $S$) checks for the emission of signal $S$, and the program preempts even before it enters the body of abort (i.e. emit $O$). Therefore, the signal $O$ is not emitted in this case.

The abort and trap statements can optionally have a handler. The handler allows programmers to implement an interrupt-handler-like callback subroutine in the SystemJ program. For example, preempting a body $p$, that is currently executing in a program abort($S$) $p$ do $q1$ ; $q2$ immediately switches control-flow to $q1$. When $p$ finishes normally (i.e. without preemption), the program continues with the execution of $q2$ rather than $q1$.

Reactions in two asynchronously running clock-domains communicate via rendezvous-style message passing over channel. A reaction in the sending clock-domain passes data over a channel using the send statement. On the other end, the data is received by a reaction in the receiving clock-domain using the receive statement. Lastly, data transferred through a channel or signal can be retrieved using the hash (#) operator.
### 2.2 The SystemJ kernel statements

Table 2.2: The SystemJ derived statements

<table>
<thead>
<tr>
<th>Statements</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>await([immediate] S)</td>
<td>abort([immediate] S){ while(true) pause; }</td>
</tr>
<tr>
<td>weak abort([immediate] S) p</td>
<td>trap(T){ {p;exit(T);}</td>
</tr>
<tr>
<td>sustain(S)</td>
<td>while(true){ emit S; pause; }</td>
</tr>
<tr>
<td>abort p</td>
<td>abort(S1){ abort(S2){ .. p .. } do q2 } do q1</td>
</tr>
<tr>
<td>case (S1) do q1</td>
<td>while(true) pause;</td>
</tr>
<tr>
<td>case (S2) do q2 ...</td>
<td>while(true) p</td>
</tr>
<tr>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>

#### 2.2.1 SystemJ derived statements

SystemJ provides a number of derived statements, which act as *syntactic sugar* to ease programming. During compilation these statements are expanded into sequences of one or more kernel or other derived statements, allowing programmers to write their programs in a more intuitive way. Table 2.2 shows how derived statements are implemented using the SystemJ base kernel statements.

Execution of a reaction can be blocked until the arrival of signal(s) using the `await` statement. For instance, `await(S); p` blocks the execution of `p` until the status of the signal `S` becomes true. It should be noted that, as shown in Table 2.2, `await` still allows the reaction to consume a logical tick (i.e. `pause`) and does not halt processing.

The `abort` statements can be qualified with the `weak` qualifier. This qualifier allows execution of their bodies for one additional tick before preemption. Consider the example shown in Figure 2.5 which demonstrates the use of the `weak` qualifier with the `abort` statement. This example is similar to that of Figure 2.4b. However, in this case, `weak abort` preempts the body after a single tick. As a result, the program emits the signal 0 but not O2.

An emitted SystemJ signal has a limited lifetime of a single tick. It is possible that the environment or the SystemJ reactions that are the recipients of the signal might not be able to capture the signal being emitted. In this case, a programmer can write the `emit` inside a SystemJ temporal loop (i.e. `while`) in order to emit the signal continuously in

```plaintext
{ 
  emit S;
  weak abort([immediate] S){
    emit 0;
    pause;
    emit 02;
  } 
}
```

Output:

```
S

O

O2
```

Figure 2.5: An effect of weak qualifier in the abort statement
each tick. Thus, the signal can be captured by others whenever they are ready. Since this is a very common programming idiom, SystemJ provides sustain as syntactic sugar, which is automatically rewritten into a while statement enclosing the signal emission. As one can see from Table 2.2 a reaction executing sustain cannot proceed further due to the infinite loop. In order to escape the loop, programmers can use one of the SystemJ preemptive statements such as abort or trap to enclose the sustain statement.

Reactive systems typically perform a specific operation upon reception of signal(s) from the environment. Hence, the ability to handle data-fusion from multiple environment sources is important for reactive languages. For example, in SystemJ, programmers can write priority-based preemption using the nested while and abort statements with a set of handler blocks. Again, SystemJ provides a corresponding derived statement, called abort-case. Note that the signal cases appearing first in abort-case have a higher priority than the ones which appear later. For example, when both signals S1 and S2 from the abort-case example in Table 2.2 become true while the program control-flow is in p, S1 has a higher priority over S2. As a result, the program preempts and jumps to the handler q1.

The halt and loop statements are also syntactic sugar, which can replace the most commonly used idioms in SystemJ programs; reactions executing halt block forever whereas loop replaces while(true).

### 2.3 An automated ice cream manufacturing facility case study

This section presents a case study: a SystemJ program which implements an automated ice cream manufacturing facility (ICMF). An ICMF is a distributed system that integrates several existing mechatronic devices and automates the manufacturing of customised ice creams, on-demand, depending upon customers’ orders. Each mechatronic device is controlled by a smart embedded controller running a clock-domain, placed across the ICMF in proximity to the mechatronic parts. An input to the ICMF is an ice-cream cone holding an ice cream, which comes with a tag attached. The arrival of an ice cream to the ICMF is detected by a tag reader, and the data read from the tag gives information about the order from a customer. The ice cream is carried to different parts of the ICMF via conveyor belts moving at constant speed. Depending on the order the ICMF changes the path of the ice cream in order to add sprinkles as toppings. The final product is placed in either of two different places in a storing station depending on the topping option. The

---

1This manufacturing system is implemented at the University of Auckland, and is controlled by distributed micro-controllers as well as custom-designed hardware units on the FPGAs.
ICMF is also capable of detecting faulty conditions on the conveyor belts and re-routing the path of an incoming ice cream. Lastly, the ICMF processes one ice cream at a time; entrance of a new ice cream is delayed until the previously entered ice cream is placed in the storing station.

Figure 2.6 shows a layout of the ICMF, which consists of conveyor belts, with photo-eyes and tag reader (PE), ice cream placement system, fault detection and recovery controller (FDRC) for object and fault detection and recovery, a turntable controller (TTC) with sprinkle dispenser, diverter controller (DC) to divert the ice cream to the correct
path and the storing station controller (SSC) that sorts ice creams to two different storing stations based on the type of toppings. The expected behaviour of the system can be described as follows: FDRC first requests the ice cream placement station to place an ice cream on the conveyor. The ice cream is then carried by the conveyor belt to the position where the PE reads a tag that is attached to the ice cream cone. If data read from this tag indicates additional toppings, the FDRC sends a message to both the DC and TTC so that they are in position to divert the ice cream to Path-2 (see Figure 2.6) by the time the ice cream reaches either one of them assuming a constant speed for the conveyor belts. Otherwise the first path (Path-1 in Figure 2.6) is taken directly to the storing station. Once the ice cream is in the storing station, the next ice cream can be manufactured. The system also needs to correctly deal with faults. For example, if Path-2 is faulty, the ice cream needs to take Path-1 to the storing station irrespective of the customers’ demands, and vice-versa.

Referring to Figure 2.7, the overall system operation of the ICMF is split into four asynchronous concurrent software behaviours encapsulated into four clock-domains (represented by rectangles), namely FDRC (fault detection and recovery controller), DC (diverter controller), TTC (turntable controller) and SSC (storing station controller). For notations used for expressing clock-domains, reactions, etc., the reader is referred to Figure 2.1. Within each clock-domain there are reactions (rounded rectangles) representing synchronous concurrent sub-behaviours. Other than detecting an incoming ice cream to the conveyor, the role of the FDRC clock-domain is also to continuously sense for faulty statuses of the two conveyor paths (Path-1 and Path-2 in Figure 2.6), which is provided by the monitoring computer (via Path1Fault and Path2Fault). In the case that both paths are faulty, FDRC stops the system thereby preventing further system malfunction. Furthermore, FDRC sends messages to DC and TTC via channels, presented with dashed rectangles, to configure the travelling path of the ice cream depending on the topping option selected by the user. TTC is responsible for controlling a mechanical turn table placing toppings on the ice cream, while DC is in charge of controlling the diverter. The SSC clock-domain groups incoming ice creams with the same toppings.

The SystemJ code implementing the ICMF is shown in Figure 2.8, where the system is composed of four different clock-domains. In SystemJ, the clock-domains are enclosed with curly brackets on the top-level of code, for example, FDRC, DC, TTC and SSC in Figure 2.8. Each clock-domain has its own set of interface signals and channels, which are declared after the clock-domain names (e.g., lines 3-6, 38-40, 57-59 and 70-72 for the clock-domains FDRC, DC, TTC and SSC, respectively). FDRC first creates an instance of a Java class called Analyser and initialises its data-structure by adding the ice-cream types that are served from this unit (lines 8-10). FDRC then requests the ice cream from the placement station through the signal PLACE (line 14). Detection of the incoming ice-cream
cone is mapped to the input signal ICDetect, which is captured by the await statement at line 15. Recall that the await statement is just syntactic sugar expanded to an abort enclosing a temporal loop (Section 2.2.1), which blocks the reaction until the presence of the signal is detected. When the ice cream is detected on the conveyor, FDRC notifies the placement station to stop placing new ice creams through signal DPLACE (line 16), and immediately starts sending a message to DC or TTC depending on the status of the conveyor belt (lines 17-28). For instance, when it is observed that the Path-1 (Figure 2.6) of the conveyor is faulty, FDRC always sends a message to DC and TTC through channels TurnR and Rotate, respectively (lines 17-19). As a result, DC emits a signal D with a value 1 (line 43), which effectively routes an ice cream cone to the Path-2, while TTC also emits a signal TurnC in order to turn the turntable (line 63) for toppings. On the other hand if Path-2 is faulty, PE only sends a message to DC through TurnL (lines 20-21), which always directs any incoming orders to the storing station without toppings (i.e. signal D with value 2 in line 49). When both the paths are faulty, FDRC clock-domain preempts the main program logic (line 12) and sustains the signal ERROR and DPLACE in order to stop the ICMF (line 34). In this case, the administrator needs to investigate and fix the faulty conveyors before resetting the system back to the normal operating mode.

```
// ---- FDRC clock - domain
FDRC{
input signal Path1Fault,Path2Fault;
input int signal ICDetect;
output signal PLACE,DPLACE,ERROR;
output boolean channel TurnR,TurnL,Rotate,DONE;->
{
  Analyser ice = new Analyser();
  ice.addType(Type.VANILLA);
  ice.addType(Type.TOPPING);
  int iteration = 0;
  abort(Path2Fault && Path1Fault) {
    while(true) {
      emit PLACE;
      await(ICDetect);
      emit DPLACE;
      present(Path1Fault && !Path2Fault){
        {send TurnR(true);}|{|send Rotate(true);}
      }
      else present(Path2Fault && !Path1Fault){
        send TurnL(true);
      }
      else present(!Path1Fault && !Path2Fault){
        if(ice.isVanilla(#ICDetect)){
          {send TurnR(true);}|{|send Rotate(true);}
        }else send TurnL(true);
      }
      receive DONE;
      System.out.println("Iter "+(iteration++));
      pause;
    }
  }
```
Depending on the path taken, it has to be made sure that the storing station controller is ready to pick up the ice cream from the correct conveyor belt. Therefore, both DC and TTC send a message to SSC through the channels Store1 and Store2 indicating that the
2.3 An automated ice cream manufacturing facility case study

ice cream is coming from Path-1 or Path-2, respectively (lines 76 and 78). Upon receiving a message from the channel, SSC emits one of the signals; enSt1 or enSt2 in order to pick up the ice cream that has reached the storing station. Finally, SSC informs FDRC about the completion of the manufacturing process via channel DONE (lines 29 and 80), which prints out the number of ice cream(s) produced to the standard output (line 30).

The example in Figure 2.8 shows a number of major features of the SystemJ language, which are:

a) Signals in SystemJ abstract away the underlying communication protocols between the reactions and the external environment. This allows programmers to focus more on actual system design rather than fiddling with low-level implementation details such as synchronisation, etc. For example, input and output signals of the clock-domain FDRC (Figure 2.8) can be as simple as digital signals via input/output pins, for controlling a mechatronic device such as ice cream placement station, or via internet protocols for exchanging information with the monitoring computer.

b) A complete system is described as a SystemJ program using the GALS paradigm. Designers are able to easily capture systems that are concurrent/parallel in nature through abstractions such as clock-domains and reactions. In addition, exchange of data between reactions in two different clock-domains over a channel does not block other reactions of the clock-domain that are not involved in the communication. This preserves clock-domain reactivity, which is crucial in designing reactive systems.

c) A behaviour of the SystemJ clock-domain or reaction can be hierarchically composed through the synchronous parallel operator || (e.g. Figure 2.8 lines 18, 25, and 77). The behaviour of spawned reactions is deterministic, following the strict synchronous MoC.

d) In SystemJ, preemption of on-going computation is well defined in the language semantics. For example, programmers can safely preempt any number of concurrently running reactions using abort, which forces the current program control-flow to jump out of the enclosing abort body. Note that this notion of preemption is different from the preemption of tasks by the operating system, which is used for temporarily interrupting the running tasks. In SystemJ, preemption only happens at the logical-level, and the clock-domains must finish executing their current tick until pause is encountered (i.e. tick is atomic).

e) Data-driven operations are expressed in Java (e.g. Figure 2.8 lines 8, 11, and 30) and can be naturally mixed with the SystemJ kernel statements. This allows programmers to design a complex system with customized data-structures and computations.
There is a significant amount of complexity and parallelism needed in the control logic of the manufacturing facility controllers, e.g., freedom from deadlocks, race conditions, etc., and there is a need to formally verify this logic to guarantee correct behaviour of the system. Furthermore, systems like ICMF need also to meet a number of real-time constraints. For instance, the ICMF cannot control the travelling speed of an ice cream on the conveyor belt, therefore generation of correct control signals to the diverter and turn-table within a time bound upon detection of an incoming ice cream, etc., need to be verified. In the following chapters, new methodologies in the SystemJ compilation process and tool-chains are introduced in order to guarantee such functional and real-time properties.

2.4 Asynchronous GRaph Code intermediate format

The original SystemJ compilation process involved four steps: 1) creating an abstract syntax tree (AST), 2) converting AST to Asynchronous GRaph Code (AGRC) [21], 3) optimising the AGRC, and 4) generating the back-end executable binary. AGRC is the primary source for the code generation of the SystemJ program, which consists of both control-dominated and data-dominated computations. It is a graph whose path from the root to the leaf node represents a tick (program transition) of the clock-domain. Each AGRC node performs a specific operation such as manipulating statuses of signals and channels, or changing the states of other nodes, thereby changing the control-flow of the program, etc. Table 2.3 gives the description of the primitive AGRC nodes, used to build the graph.

The AGRC node is said to be activated when the control-flow of a program triggers an input port go. Once a node is activated, the corresponding operation is performed depending on the type of a node. One or more output ports out are then triggered, which consequently activates successive children nodes connected to those output ports.

Every AGRC constructed from the SystemJ program has a root node called the async fork node which starts one or more clock-domains simultaneously. Control-flow of each clock-domain is determined by the switch and test nodes. The switch node is like a state variable of the clock-domain, and a set of values of all switch nodes in the AGRC represents the current state of the SystemJ program. The value of switch nodes is manipulated by enter nodes during execution of a clock-domain tick. Signal statuses or conditional expressions used in the program are checked in the test nodes. Any data computations or signal emissions are done in the action nodes. SystemJ reactions are forked by the fork node where all the children nodes connected through the output ports of the fork node (i.e. C0 to Cn, in Table 2.3) are activated simultaneously. The terminate node is used to
### 2.4 Asynchronous GRaph Code intermediate format

#### Table 2.3: The semantics of the AGRC nodes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Node name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Action node diagram" /></td>
<td>Action node</td>
<td>Control-flow reaching the input port <code>go</code> of this node will activate the action(s) and instantaneously trigger <code>out</code>. The actions may consist of signal emissions and data computations.</td>
</tr>
<tr>
<td><img src="image" alt="Enter node diagram" /></td>
<td>Enter node</td>
<td>When <code>go</code> is triggered, the enter node changes the state of the switch node to <code>value</code>.</td>
</tr>
<tr>
<td><img src="image" alt="Switch node diagram" /></td>
<td>Switch node</td>
<td>Depends on the <code>state</code> encoded from the enter node, the switch node triggers one of its children (<code>Cx</code>). The set of switch node values in AGRC represents the overall state of a SystemJ program. Each switch node is associated with at least one enter node that encodes its state.</td>
</tr>
<tr>
<td><img src="image" alt="Test node diagram" /></td>
<td>Test node</td>
<td>The test node evaluates the assigned expression, and triggers one of two branches depending on the result. The node is used for testing the presence of the signals or conditional expressions for branches. Depending on the result, one of <code>then</code> or <code>else</code> output branch (port) is triggered.</td>
</tr>
<tr>
<td><img src="image" alt="Fork node diagram" /></td>
<td>Fork node</td>
<td>SystemJ reactions are forked at the fork node. The number of output ports indicates the number of forked reactions. Unlike the switch node, the fork node triggers all output ports when control-flow activates <code>go</code>.</td>
</tr>
<tr>
<td><img src="image" alt="Terminate node diagram" /></td>
<td>Terminate node</td>
<td>The terminate node indicates the exit status of the SystemJ processes (clock-domains or reactions). The exit code of zero indicates termination of the process whereas the exit code of one indicates a completion of one logical tick. Any exit codes above one indicate software preemption (i.e. <code>exit(T)</code>).</td>
</tr>
<tr>
<td><img src="image" alt="Join node diagram" /></td>
<td>Join node</td>
<td>The join node acts as a synchroniser which joins all forked reactions. The input ports of the join node represent all possible exit codes from the forked reactions. The output port <code>Cn</code> is activated where ( n = \text{MAX}(k_0, \ldots, k_n) ).</td>
</tr>
<tr>
<td><img src="image" alt="Async fork diagram" /></td>
<td>Async fork</td>
<td>Clock-domains are forked at the async fork node. The Async fork node is the root node of AGRC.</td>
</tr>
<tr>
<td><img src="image" alt="Async join diagram" /></td>
<td>Async join</td>
<td>The Async join node joins clock-domains. Unlike the join node, the async join does not synchronise clock-domains.</td>
</tr>
</tbody>
</table>
determine the exit statuses of the clock-domains or reactions, and decides how the program control-flow should proceed. For example, the join node activates one of the output ports depending upon the largest incoming exit code generated by the forked reactions. The control-flow cannot proceed further until all the forked reactions are synchronised at the join node. Lastly, the async join node groups the clock-domains forked at the async fork node, however unlike reactions, clock-domains do not wait at the asynch join node as they are asynchronous to each other. Table 2.3 gives more detailed description of these AGRC nodes.

The AGRC graph for the storing station controller (SSC) from Figure 2.8 is shown in Figure 2.9. Note that some parts of the graph, which implement the channel communications and the other clock-domains (i.e. FDRC, DC and TTC), are abstracted to a single node (rectangular box) for sake of simplicity. Traversing the graph from the root node (async fork) to the leaf node (async join) implies execution of a single logical tick. Furthermore, suppose the value of all the switch nodes is initialised to 1 (i.e. triggers the child branch
The program starts from the async fork node (AFK), which starts SSC and all other clock-domains simultaneously by triggering all of its output ports. When the control-flow reaches S1, it decodes the switch node’s state as 1 and enters the corresponding bottom branch. The weak abort (line 75 in Figure 2.8) spawns two synchronous reactions at the fork node F1, as according to the expansion rule shown in Table 2.2. Two enter nodes then encode S1 and S2 to 0. Note that the statuses of enSt1 and enSt2 are not checked during the first tick (i.e. branch 1 of S1) since the abort is non-immediate. Next, the program forks two additional reactions at the fork node F2 (line 77 in Figure 2.8) and starts the channel communications in each of these reactions via Store1 and Store2. The channel communication may not be completed within a single tick, in such case, the control-flow takes ‘Not-fin’ branch and finishes a tick with a termination code 1. Upon completion of the channel communications, the signals enSt1 and enSt2 are emitted from each of the reactions, respectively. The join node J2 computes the maximum terminate code from the reactions forked at F2 and triggers corresponding branch (0 or 1). On the other hand, the reactions forked by the weak abort at F3 or F1 are joined at J1. Note that sharing a single join node (J1) for 2 fork nodes (F1, F3) is an optimisation. The same rule is applied to J1 for computing the maximum termination code.

When all the reactions finish with termination codes of 0, the enter node encodes S2 to 1 (i.e. left branch of J1) and the program immediately starts another channel communication in the same tick via DONE. Upon its completion, the program re-iterates the loop (line 74 in Figure 2.8), by reinitialising both S1 and S2 to 1. The left branch of J1 is also taken, which also starts the channel communication over DONE, when the expression enSt1 || enSt2 tested in the node T1 evaluates to true. This behaviour can be easily observed at lines 75-80 in Figure 2.8 where send DONE is performed after aborting two reactions spawned in the body of abort.

2.5 Causal cycles

The original SystemJ’s signal broadcast communication model, which is based on instantaneous broadcasting of signals (status and value) and their propagation across the synchronous reactions (Section 2.1), exhibits some semantic challenges which will be discussed in this section. As a result, SystemJ programs can be written in such a way that their execution behaviours are logically incoherent or non-reactive. In [50], Berry defines coherency of purely synchronous programs with respect to each signal as follows:

*The global status of the program is logically coherent iff at least one emit statement is executed for each signal assumed present and no emit statement is executed for each signal assumed absent.*
During execution of any tick, when this property is violated, the program is considered having a causal error and the compiler normally rejects such programs. For example, consider the SystemJ code below:

```plaintext
{  
  signal O; // Signal declaration  
  present(O){ /* Do nothing */ } // branch-1  
  else { emit O; } // branch-2  
} // Program terminates
```

At the time of execution of the `present` statement, the control-flow enters `branch-1` when the status of the signal `O` is true, otherwise it enters `branch-2`. However, further execution of the program invalidates the previous assumptions since: 1) the signal `O` is not emitted in `branch-1` and 2) the signal `O` is emitted in `branch-2`.

By slightly modifying the previous example, the program can even be non-deterministic:

```plaintext
{  
  signal O; // Signal declaration  
  present(O){ emit O; } // branch-1  
  else { /* Do nothing */ } // branch-2  
} // Program terminates
```

In this example, both branches are logically coherent, and executing this program results in non-deterministic behaviour, i.e. one cannot determine whether or not this program will emit the signal `O`. Causality problems can also occur due to valued signal emissions:

```plaintext
{ emit S(#S + 1) }
```

The program above tries to emit the signal `S` with a value of `#S + 1`. However, this creates an infinite cycle in determining the value of `S`. For example, assume the value of `S` is 0 prior to execution of `emit` statement, then the argument of `emit` is evaluated to `0 + 1 = 1`. However, the previous assumption on `#S`, which is 0, is now invalidated since the value of `S` is now 1, hence the causal error.

Berry [50] states that synchronous programs having causal cycles are logically incorrect if they are incoherent or non-deterministic. This is also true for SystemJ clock-domains as synchronous semantics is a subset of the GALS MoC. However, causal programs are not always incorrect. In fact, programmers can still write a correct program, which contains causal cycles. Consider the following example:

```plaintext
{  
  signal A,B1,B2,O1,O2;  
  emit A;  
  present(B2) emit O1; else emit O2; // First present  
  present(A) emit B1; else emit B2; // Second present
```
The status of signal B2 cannot be determined by the time the control-flow reaches the first present statement. It is because the potential emission of B2 can only be ruled out depending on the result of the second present statement. In order to resolve this problem, some compilers perform a sort of forward evaluation of signals through a mechanism called potential computation [50]. This is used to determine statuses of signals in the current statement on the basis of constructive provability. For instance, Esterel v5.92 compiler [51] accepts the above program, which is cyclic but constructive.

The mechanism of instantaneous signal broadcasting makes analysing timing properties of SystemJ programs more difficult. This is mainly due to the current implementation of determining the status of the signals at each tick. For example, consider the example below:

```plaintext
{signal A, B, C;
  /* R1 */ /* R2 */ /* R3 */
  {present(A){emit B;}}||{present(B){emit C;}}||{emit A;}
}
```

According to the SystemJ semantics, the signals A, B and C are all emitted as a result of execution of this program; the emission of the signal A in reaction R3 leads to reaction R1 emitting B followed by emission of signal C in reaction R2. The SystemJ compiler generates an executable from this program using the cyclic scheduling policy [21]. One can use the cyclic scheduling policy to extract real parallelism on multi-processor system by running the concurrent reactions individually, one after another. Therefore, status of signals can be unknown at the time when the program control-flow reaches the signal check statement (e.g. present) until statuses of all signals, which can be potentially emitted in other reactions, are resolved. In order to resolve the cyclic signal dependencies among the concurrent reactions, the SystemJ program maintains, for each signal, an additional hidden state called unknown. Whenever the control-flow tries to check the signal whose status is unknown, it locks the execution of the currently running reaction and executes the next reaction, if any. The locked reaction is revisited after running all other reactions and this is repeated until all signal statuses are resolved.

The cyclic scheduling policy is a run-time method for resolving the status of signals. Therefore, this approach is not favoured for analysing timing properties, particularly the worst case reaction time, of a clock-domain. For instance, consider the previously shown example having signal dependencies (R1 to R3). Assuming that the generated program runs reactions from R1 to R3, it is required to run R1 and R2 twice in order to resolve the status of the signals A and B. However, the language semantics does not enforce the execution order of reactions; it does not prevent the compiler from generating code,
which executes the reactions in the order: R3, R1, and R2. In this case, all the signals are resolved at the time of first execution of each reaction. Furthermore, depending on the implementation, reactions can even be run on different resources (i.e. processors) in order to exploit real parallelism, in which case, the order may be unknown.

To overcome the aforementioned problems, a new signal communication model is introduced in the SystemJ language. In this model, SystemJ programs always check for internally emitted signal statuses from the previous tick instance. By delaying reactions to internally emitted signals by one tick, more programs can be accepted by the compiler without need for complex analysis techniques (e.g. potential computation). As it will be explained in the next chapter, all the incoherent and non-deterministic programs presented in this section are accepted and become coherent and deterministic under the new signal broadcast model.
Semantics of the SC-SystemJ Language

This chapter presents Safety-Critical SystemJ (SC-SystemJ), which defines a subset of the original SystemJ language suitable for verifying functional correctness of safety-critical systems. In particular, the semantics of SC-SystemJ is based on automata theory whereby each clock-domain is represented as a finite state machine (FSM). Section 3.1 presents the motivation of SC-SystemJ for designing safety-critical systems. Section 3.2 describes the SC-SystemJ kernel as well as derived statements and show the differences compared with the original SystemJ language. Section 3.3 introduces a new signal communication model for synchronous reactions within a clock-domain. A brief introduction on Linear Temporal Logic (LTL) is given in Section 3.4 as background to assist with the understanding of the transition semantics of the SC-SystemJ program described in Section 3.5.

3.1 Motivation behind introducing SC-SystemJ

One of the major features of the SystemJ language is the ability to perform data-computations using Java. This appeals to many traditional software developers who are already familiar with imperative and object-oriented programming concepts. For example, developers are allowed to implement complex data-structures in the program using full-fledged Java’s object-oriented programming model as well as the large set of standard libraries. Since SystemJ programs are compiled into Java source code, all the data-oriented computations
Semantics of the SC-SystemJ Language

described in Java can also be easily included in the back-end code for execution. While such features make the SystemJ language very flexible, enabling programmers to target a wide range of application domains, it is not well suited for designing safety-critical systems because:

1) *Garbage collection (GC)* is a necessity in the object-oriented programming model of Java. Typical data-computations described in Java often require allocating memory at runtime for newly created objects. GC is an automatic memory management process which de-allocates memory space occupied by objects when they are no longer used by the program. It is a well-known problem that a tight bound of the worst-case GC cycle time is difficult to find, and the results are largely overestimated [52].

2) *Conflicts with the SystemJ semantics* and the Java threading model makes SystemJ programs hard to analyse. For example, SystemJ programmers are able to create Java threads at runtime, and even perform inter-process communication with SystemJ clock-domains and reactions when necessary. Yet, there is no semantic rule defining how SystemJ clock-domains or reactions should run and interact with the Java threads. Such semantic ambiguity introduced by Java is obviously unwanted in the verification of safety-critical SystemJ programs.

To address the aforementioned issues, a subset of the SystemJ language called SC-SystemJ is introduced in this chapter which restricts the use of Java constructs in the original language. More precisely, SC-SystemJ does not allow the creation of any Java objects in the program, and only supports primitive data types such as `int` and `short` for data-oriented computations. This avoids the need for triggering GC and also prohibits creating Java threads at runtime, making SC-SystemJ suitable for developing safety-critical applications.

As already described in [Section 2.4](#), the micro-step semantics of the SystemJ language gives a precise view of how a program executes, in the presence of GALS concurrent processes (unlike Java threads), by the definition of a sequence of elementary nodes for every SystemJ kernel statement. In this regard, the micro-step based semantics, which constructs the AGRC as an intermediate format, is particularly useful for reasoning about a program’s behaviour for each clock-domain tick. Nevertheless, one cannot easily correlate the program state encoded in the AGRC, represented as a set of switch nodes (as described in [Section 2.4](#)), with the exact location in the SystemJ program. This makes the formulation of correctness properties particularly difficult in the verification process. Moreover, the AGRC can potentially have a large number of states which can increase the verification time quite significantly. To address these problems, SC-SystemJ adopts big-step semantics that transforms the program into a *network of FSMs*, where each clock-
domain is represented as an FSM. SC-SystemJ encodes FSM states only with the `pause` statements within a clock-domain and the state transition is simply advancement of a tick from one `pause` to the next. Therefore, every state in the FSM has a direct correlation with the `pause` statement(s) in the program. Furthermore, any instantaneous statements cannot create states, thereby reducing the total number of states required to represent the program.

Rendezvous communication in the original SystemJ language is implemented using a mixture of control and data-oriented computations. Moreover, channels consist of a number of status variables, which are encoded using an integer type. As a result, the complexity of verifying a program using channels would increase due to the required amount of data computations in the integer domain. In SC-SystemJ, a more transparent and easily verifiable approach is adopted for the channel implementation using the four-phase handshake mechanism \[53, 54\].

As mentioned previously, an intermediate format of the SC-SystemJ program is a network of FSMs. A primary reason for generating the FSMs is to take advantage of many available and well-established techniques which can be used to verify systems modelled in one or more FSMs. In particular, this thesis employs model checking \[55\] and Satisfiability Modulo Theories (SMT) \[46\] techniques for functional and real-time verification, respectively. A procedure for generating the FSMs involves several steps, which require many intricate details. In this chapter, control-flow and data-flow semantics rules are first presented. These rules are applied inductively to each SC-SystemJ clock-domain to generate transition systems describing behaviours of the clock-domains. A GALS system is created from the asynchronous composition of these transition systems. As will be shown in Chapter 4, these transition systems are transformed into a network of FSMs for verification.

### 3.2 Syntax for the SC-SystemJ statements

The syntax of the synchronous part of the SC-SystemJ (also SystemJ) language is similar to Esterel. Additionally SC-SystemJ has `send` and `receive` statements that are used for synchronising and exchanging data between reactions in different clock-domains.

Since SC-SystemJ is a subset of SystemJ, most of its kernel statements are analogous to the original version with few exceptions. Hence, SystemJ programmers would be able to easily adapt to the SC-SystemJ programming style without much effort. Table 3.1 shows a complete list of the SC-SystemJ kernel statements in comparison with the corresponding SystemJ syntax. Signal and channel declaration statements are declared in the same fashion as original SystemJ. In SC-SystemJ however, signals or channel values can be
<table>
<thead>
<tr>
<th>SC-SJ statement</th>
<th>SJ statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[input/output] [type] signal S</td>
<td>[input/output] [type] signal S</td>
<td>Declaring a pure or valued signal, optionally with an associative and commutative operator.</td>
</tr>
<tr>
<td>[op (+</td>
<td>*) = n]</td>
<td></td>
</tr>
<tr>
<td>[input/output] [type] channel C</td>
<td>[input/output] [type] channel C</td>
<td>Declaring a channel with optionally a value initialised to n.</td>
</tr>
<tr>
<td>[L:] pause</td>
<td>pause</td>
<td>Labelled pause that consumes a logical instant of time.</td>
</tr>
<tr>
<td>emit S</td>
<td>identical</td>
<td>Emitting a signal for one tick</td>
</tr>
<tr>
<td>#S = exp</td>
<td>emit S(exp)</td>
<td>Assigning a result of exp to a signal S</td>
</tr>
<tr>
<td>while(true) p</td>
<td>identical</td>
<td>Temporal loop</td>
</tr>
<tr>
<td>present(S) p1 else p2</td>
<td>identical</td>
<td>If signal S is present do p1 else do p2</td>
</tr>
<tr>
<td>if(exp) p1 else p2</td>
<td></td>
<td>If the expression exp is evaluated to true, do p2 else do p2</td>
</tr>
<tr>
<td>abort(S) p</td>
<td>abort([immediate] S) [do q]</td>
<td>Preempt p if S is present</td>
</tr>
<tr>
<td>suspend(S) p</td>
<td>suspend([immediate] S) p</td>
<td>suspend for 1 tick if S is present</td>
</tr>
<tr>
<td>—</td>
<td>trap (T) p [do q]</td>
<td>Software exception preempting p</td>
</tr>
<tr>
<td>—</td>
<td>exit(T)</td>
<td>Throw a software exception preempting p</td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>send C[[exp]]</td>
<td>identical</td>
<td>Sending data over the channel</td>
</tr>
<tr>
<td>receive C</td>
<td>identical</td>
<td>Receiving data over the channel</td>
</tr>
<tr>
<td>#C</td>
<td>identical</td>
<td>Retrieving data from a valued signal or channel</td>
</tr>
<tr>
<td>—</td>
<td>jterm(p)</td>
<td>Java data-oriented computation</td>
</tr>
<tr>
<td>extern func([arglist])</td>
<td>—</td>
<td>Calling an external function</td>
</tr>
</tbody>
</table>

initialised at their declaration. Additionally, signals can be declared with an associative and commutative operator $\text{op}$, which is used to combine multiple emissions of the same valued signal in the same tick. The $\text{pause}$ statement can optionally be labelled with an arbitrary token L. As will be seen in the next chapter, these labels signify explicit states in the SC-SystemJ program. A signal value is now assigned with the hash operator (#) in SC-SystemJ whereas it is done using the $\text{emit}$ statement in SystemJ. The $\text{abort}$ and $\text{suspend}$ statements with immediate qualifier are now syntactic sugar in SC-SystemJ as shown in Table 3.2. Although the $\text{trap}$ statement is omitted as a kernel statement, it can be emulated with $\text{abort}$ statements. Unlike SystemJ, SC-SystemJ provides the $\text{if-else}$ statement as a kernel, which is used to branch the program flow depending on the result of the data expression exp. As explained in Section 3.1, use of Java computation is restricted in SC-SystemJ, and they cannot be directly used in the SC-SystemJ program. If desired programmers can still describe full-fledged Java-based data operations inside the $\text{extern}$ function call. However, verification of the program logic using $\text{extern}$ function call is not accommodated in this work, and left as future work. The Extended Backus-Naur Form
3.3 Delayed signal semantics

The first implementation of the original SystemJ was based on the zero delay model where the program reacts instantaneously to any events on signals; both I/O and internal signals, within the current tick. Although such a model gives an opportunity for programmers to build a completely combinational (memoryless) system, which is beneficial in some situations, there are some shortcomings:

a) Instantaneous broadcast and reaction to signals imposes compile-time or run-time overheads for resolving all causal cycles, which are induced from signal communications within or amongst concurrent reactions for each clock-domain tick. As discussed in Section 2.5, the original SystemJ compiler uses the run-time approach for resolving the signal dependencies across concurrent reactions as this approach allows extracting parallelism on multi-core systems. However, this approach hinders real-time analysis of the SystemJ programs, particularly WCRT, because the time for resolving these causal dependencies cannot be known a priori at compile time.

b) Detecting causal errors in the SC-SystemJ’s automata-based semantics is more difficult than the original semantics described in [21]. It is mainly because the new semantics is based on big-step execution of clock-domains, which is unconcerned about how signals are propagated in the program transitions.

Therefore, SystemJ’s signal communication model is relaxed by forcing SystemJ clock-domains to react on any emitted signals in the following tick. This change also has been incorporated in the signal communication model of SC-SystemJ. Accordingly, any signal communication among synchronous reactions is effectively delayed by one tick.

Figure 3.1a shows an example program with signal traces for four consecutive ticks (T1...T4) when executed under the original instantaneous and the new delayed signal communication semantics. The program is a temporal loop, enclosing the present statement, which is emitting signal $B$ or signal $C$ depending on the status of the signal $A$. Additionally, signal $A$ is emitted and checked in the same tick by emit and present statements, respectively, and there is no pause between these statements. Under the original semantics, emission of $A$ is instantaneously checked by the present statement, which consequently results in the immediate emission of $B$. Therefore, the signal trace in Figure 3.1b shows that the statuses for both $A$ and $B$ are set to true for all ticks. On the other hand, under delayed semantics, presence check for the signal $A$ is delayed by one tick, which
signal A;
while(true){
    emit A;
    present(A){
        emit B;
    } else {
        emit C;
    }
pause;
}

(a) A (SC-)SystemJ program emitting and capturing internal signals

(b) Execution trace for instantaneous broadcast communication model

(c) Execution trace for delayed broadcast communication model

Figure 3.1: Different execution traces for instantaneous and delayed signal semantics

leads the control-flow to take the else branch of present statement and emit C at end of tick T1. As a result, both A and C are set to high in T1. Finally, the signal A emitted in T1 is captured by the present statement in T2, which leads to the emission of B in the same tick (shown in Figure 3.1c).

As can be seen from the previous example, delayed semantics changes the behaviour of the program quite significantly. In particular, when an input to the system triggers a chain of internal signal communications, the corresponding output response can be delayed by as much as the length of this chain. Nevertheless, delayed signal semantics removes causality and cyclic dependency problems; control-flow of a program does not change with respect to the status or value of the signals emitted during that tick. This is not the case for the original semantics where the program needs to react to the signals, which are emitted during the current tick. For instance, recall the incorrect SystemJ program present(0) else emit 0; shown previously in Section 2.5. When the emission of signal 0 is delayed by one tick, control-flow can take the else branch without causal error since its status will remain false until the end of the current tick. Likewise, the program present(0) emit 0; else ; shown in Section 2.5 becomes deterministic under delayed signal semantics. In addition, delayed semantics also applies to the signal values; any signal value emitted can only be retrieved in the next tick. Therefore, there can no longer be any causal error
3.3 Delayed signal semantics

in valued signal emissions, such as $\text{emit } S(#S + 1)$ shown in Section 2.5. Lastly, since the statuses of the emitted signals in the previous instant are known at the end of each tick, extra complexity introduced for resolving those signals is no longer needed, making the program more amenable for real-time analysis.

In Esterel, there is a $\text{pre}$ operator such that $\text{pre}(S)$ gives the status of the signal $S$ in the previous instant [56]. However, SC-SystemJ’s (and also SystemJ’s) delayed signal semantics is different from $\text{pre}$ defined in [56], which is expanded to kernel statements as shown in the following ($p$ is a program in the scope of the signal $S$):

```plaintext
trap(T){
    signal S, preS;
    { p ; exit(T); } ||
    {
        while(true){
            present(S){ pause; emit preS; }
            else pause;
        }
    }
}
```

Here, an additional signal called $\text{preS}$ is declared, which is a delayed version of the signal $S$. For every emission of $S$ in $p$, $\text{preS}$ is also emitted in the next tick. Every $\text{pre}(S)$ occurring in $p$ is then replaced with $\text{preS}$, which effectively makes the program react to $S$ with a single tick delay. One drawback of this ‘syntactic sugaring’ approach, which is also mentioned in [56], is that of increasing compiled code size due to the additional program logic. On the other hand, SC-SystemJ implements the signal delay at the core language level, rather than as a derived statement.

Reading status of communicating signals in delayed fashion have been adopted in designing heterogeneous systems in which multiple hardware IP blocks operating at distinct clocks are integrated via Network on Chip (NoC) communication fabric. For example, authors in [57] presented modelling of an NoC architecture including its communication scheme using a formal language called Heterogeneous Protocol Automata (HPA). Similar to SC-SystemJ, automata in a HPA system communicate via channels. In addition, HPA supports reading statuses of signals on these channels in the previous clock cycle, providing flexibility in designing the communication protocol between processing modules in a system.

In the next section, a brief introduction on Linear Temporal Logic (LTL) [43] is presented in order to help readers understand the semantics of the SC-SystemJ language. A transition system, derived from a SC-SystemJ program, is expressed symbolically as
an LTL formula. Each SC-SystemJ control statement, presented in Section 3.2, is translated into an LTL formula using well-defined inductive rules, which will be described in Section 3.5. All formulas are then combined together, resulting in a transition system for the whole SC-SystemJ program. The resultant transition system can be verified against a set of properties, which are also described in LTL.

### 3.4 Linear temporal logic

*Linear Temporal Logic* (LTL) is a temporal logic [43], which has been used for many years as a language to build a formal specification of concurrent programs. Temporal logic like LTL consists of a set of propositions, for example ‘the program is in an idle state’, which can be qualified in terms of time, for example ‘the program will always eventually go out of the idle state’. One or more propositions combined with usual boolean operators (e.g. conjunction and disjunction) form an LTL formula, which is used for establishing formal reasoning about the concurrent programs represented as infinite sequences of states. Such LTL formulas are used by various types of verification tools, such as model-checkers, in order to verify the program (system) properties such as deadlock freedom, freedom from starvation etc.

Formally, LTL is evaluated over the infinite sequence of states (or labels) \( \pi = q_0, q_1, q_2, \ldots \), where for all \( i \), there exists a transition \( q_i \rightarrow q_{i+1} \) and \( \pi^i \) is the suffix of \( \pi \) starting from the \( i \)th state, i.e. \( q_i, q_{i+1}, \ldots \). A well-formed LTL formula consists of atomic propositions from a finite set \( P \), generally specified in alphabets \( x, y \), or \( p \) etc., boolean (logical) operators, such as *and* (\( \land \)), *or* (\( \lor \)), and *not* (\( \neg \)), and temporal operators for describing time, such as *next* (\( X \)) and *until* (\( U \)). When every atomic proposition \( p \in P \) is an LTL formula, then all of the following are also LTL formulas: \( \neg p \), \( p_1 \land p_2 \), \( X p \) and \( p_1 U p_2 \). An interpretation for an LTL formula is a function \( \pi : N \rightarrow 2^P \), which maps each instant of time into a set of propositions that hold in that instant. Semantics of LTL is then further defined as follows:

\[
\begin{align*}
\pi \models p & \quad \text{iff} \quad p \in q_0, \ p \in P \\
\pi \models \neg p & \quad \text{iff} \quad \pi \not\models p \\
\pi \models p_1 \lor p_2 & \quad \text{iff} \quad \pi \models p_1 \text{ or } \pi \models p_2 \\
\pi \models X p & \quad \text{iff} \quad \pi^1 \models p \\
\pi \models p_1 U p_2 & \quad \text{iff} \quad \exists i \geq 0 \text{ s.t. } \pi^i \models p_2 \text{ and } \pi^j \models p_1, \forall j, 0 \leq j < i.
\end{align*}
\]

For a given infinite sequence of states \( \pi \), a proposition \( p \) is true if and only if \( p \in \)
3.4 Linear temporal logic

$q_0$, where $p \in P$. In this case it is said $\pi$ entails ($\models p$). The boolean operators $\lor$ and $\neg$ imply logical disjunction and negation, respectively. For example, LTL formula $p_1 \lor p_2$ holds for an infinite sequence of states $\pi$ when either one of the propositions $p_1$ or $p_2$ hold at $q_0$. Also $\neg p$ holds for $\pi$ when $p$ does not hold at $q_0$. Temporal operator $Xp$ states that the proposition $p$ holds in the next state. On the other hand $p_1U p_2$ states that the proposition $p_1$ holds from $q_0$ until $p_2$ holds in some $q_j$ where $j \geq 0$. Graphical illustrations of these temporal operators are shown in Figure 3.2.

There are also a number of other LTL operators which can be derived from the kernel LTL operators shown above:

\[
\begin{align*}
\top & \equiv p \lor \neg p \\
\bot & \equiv \neg \top \\
p_1 \land p_2 & \equiv \neg (\neg p_1 \lor \neg p_2) \\
Fp & \equiv \top U p \\
Gp & \equiv \neg F \neg p \\
p_1 \implies p_2 & \equiv \neg p_1 \lor p_2 \\
p_1 \iff p_2 & \equiv (p_1 \implies p_2) \land (p_2 \implies p_1)
\end{align*}
\]

The boolean true and false operators are denoted by $\top$ and $\bot$, respectively. The logical conjunction ($\land$) is constructed using combinations of logical negations ($\neg$) and a logical disjunction ($\lor$). Two additional temporal operators $F$ and $G$ are also frequently used. For example, $Fp$ states that the proposition $p$ will eventually hold in the future, whereas $Gp$ states that $p$ holds globally. Figure 3.3 shows two infinite paths, which illustrate these temporal operators. Lastly, operators $\implies$ and $\iff$ are used for logical implication and equivalence, respectively.

![Figure 3.2: Two infinite sequences of states that hold $Xp$ and $p_1U p_2$](image-url)
3.5 Transition semantics of SC-SystemJ programs

This thesis employs LTL for two major reasons: (1) to build a transition system of SC-SystemJ clock-domains (2) to verify functional correctness of SC-SystemJ programs. Details of the verification procedure are presented in the next chapter. In order to verify SC-SystemJ programs however, a transition system first needs to be derived, which is a formal model used for describing the behaviour of SC-SystemJ clock-domains. In this thesis, the transition system of a clock-domain is derived inductively using the control-flow and data-flow semantics of the SC-SystemJ statements. These semantics are described in LTL. Therefore, the resultant transition system is also an LTL formula.

Definition 1. Transition system: Every SC-SystemJ clock-domain is a mealy machine \((Q, st, I, O, Y, A, T)\) where:

- \(Q\) is the set of states
- \(st \in Q\) is the starting state
- \(I\) is the set of input signals
- \(O\) is the set of output signals
- \(Y\) is the set of internal signals
- \(A\) is the set of actions (e.g. data computations and signal emissions).
- \(T\) is the transition relation: \(T \subseteq Q \times \mathcal{B}(I \cup O \cup Y) \times 2^O \times 2^Y \times 2^A \times Q\). \(\mathcal{B}(i)\) is a Boolean expression over the symbols in \(I \cup O \cup Y\).

Simply put, a clock-domain is a directed graph with edges carrying the labels of the form \(b/A', O', Y' : b \in \mathcal{B}(I \cup O \cup Y), A' \subseteq A, O' \subseteq O, Y' \subseteq Y\). Intuitively, each edge can be taken if the Boolean condition on the edge holds true. Furthermore, actions (functions) are performed and output signals emitted upon taking the transition.
3.5 Transition semantics of SC-SystemJ programs

The system defined above consists of all the states of the clock-domain as well as the transition relations between these states. The generation of this system, which will be presented in the following section, is inductively derived from the SC-SystemJ statements. The resultant transition system is a single LTL formula which describes all possible executions of a SC-SystemJ clock-domain. On the other hand, the asynchronous composition of clock-domains is given in terms of a transition system (rather than giving it inductively), because, although an inductive definition of the asynchronous composition is possible, the result is a single automaton which can no longer be distributed across different machines to extract parallelism. Furthermore, as will be presented in the next chapter, well-developed model-checking tools, such as SPIN [44], already provide efficient asynchronous composition of a network of automata. Hence, building such a network suffices for both verification and code generation, which will be also presented in the next chapter.

The transition semantics of the SC-SystemJ language is inspired by [58]. In their work, Schneider gives an inductive definition of the transition semantics of the purely synchronous language called Quartz [28]. In this thesis, his definitions of the Quartz language have been extended (and in some places corrected) for GALS systems with the addition of delayed signal semantics as well as new data-flow semantics. Moreover, while transition system of Quartz is used for proving properties of Quartz via interactive theorem provers, this thesis focuses on WYPIWYE program compilation.

3.5.1 Transition semantics of synchronous clock-domain

Every SC-SystemJ transition system consists of control-flow and data-flow. For ease of understanding, the control-flow semantics and the data-flow semantics are described separately.

Control-flow semantics

The \texttt{pause} construct is the only construct in SC-SystemJ programs to demarcate the end of a logical tick and also represent the states of the program. Every \texttt{pause} construct in the SC-SystemJ program is labelled either by the programmer or internally by the compiler. This label on the \texttt{pause} construct indicates the state of the program. For example, the program is \texttt{in} some state \( l \) if the control-flow is \texttt{in} a statement with label \( l \). Formally, all the reachable states in some statement \( S \) are defined as:

\[
in(S) := \bigvee_{l \in \text{labels}(S)} l
\]  

Additionally, there are four types of control-flow predicates, which are used to describe the state of execution of SC-SystemJ statement \( S \):
Definition 2. **Inst(S):** Given a SC-SystemJ statement S, Inst(S) defines all the conditions where the control-flow instantaneously terminates.

Informally, control-flow never encounters pause inside S.

Definition 3. **Enter(S):** Given a SC-SystemJ statement S, Enter(S) defines all the conditions where the control-flow enters S.

Informally, control-flow encounters one or more pause statements inside S.

Definition 4. **Term(S):** Given a SC-SystemJ statement S, Term(S) defines all the conditions where the control-flow terminates (exits) S.

Informally, control-flow leaves all pause constructs inside S.

Definition 5. **Move(S):** Given a SC-SystemJ statement S, Move(S) defines all the conditions where the control-flow moves from some state inside S to another state inside S.

Informally, control-flow moves from one or more pause statements in S to another pause(s) in S. Finally, there is also the possibility that the control-flow loops on the same state(s). This is called stutter. Formally, stutter is defined as follows (X is the next-time operator):

\[
\text{stutter}(S) := \bigwedge_{l \in \text{labels}(S)} (l \iff Xl) \quad (3.2)
\]

For a SC-SystemJ statement S, given \(st \implies \text{Term}(S) \lor \neg \text{in}(S)\) holds, the transition system \(T(st, S)\) is defined as follows in the disjunctive normal form (DNF):

\[
T(st, S) := \begin{cases} 
(st \wedge \text{Inst}(S) \wedge \neg \text{in}(S)) \lor \\
(st \wedge \text{Enter}(S)) \lor \\
(\neg st \land \neg \text{in}(S)) \lor \\
\text{Move}(S)
\end{cases} \quad (3.3)
\]

For a given SC-SystemJ statement S, Equation 3.3 is applied inductively until its disjunctive normal form contains no control-flow predicates. The corresponding Mealy machine (Definition 1) is then built from this formula using the algorithm which will be explained in the next chapter. Note that Equation 3.3 is only valid under the assumption that in the starting state \(st\), the program is not running. Furthermore, in Equation 3.3 the four clauses indicate:

1) **Instantaneous transition:** The program S can make an instantaneous transition from \(st\) if there are no pause statements in the clock-domain.
3.5 Transition semantics of SC-SystemJ programs

The inductive definitions for the SC-SystemJ statements are shown in Figure 3.4 and Figure 3.5. The conditions where the SC-SystemJ statement can be instantaneously terminated are defined by \( \text{Inst} \) as shown in Figure 3.4a. When \( \text{Inst} \) is applied to a signal or channel declaration, the inductive definition evaluates them to \( \top \), which means true. It is trivial to see this since it is already known that \text{pause} is the only SC-SystemJ statement which consumes a logical tick. \( \text{Inst} \) of the signal emission statement \text{emit} \( \gamma \) gives \( X\gamma \), which means that the proposition (i.e. emission of the signal) holds in the next state. On the other hand, because \text{pause} cannot terminate instantaneously, its induction rule gives \( \bot \). The \( \text{Inst} \) of the \text{present} statement is defined as the disjunction of two conjunctive clauses since it is always the case that only one of the clauses will be evaluated to \( \top \). The

2) Entering transition: The program can settle at the first \text{pause} encountered in the program.

3) Terminate transition: The program can terminate.

4) Move transition: The program can move from some \text{pause} statement inside \( S \) to some other \text{pause} statement inside \( S \).
Semantic of the SC-SystemJ Language

Figure 3.5: Inductive definitions for SC-SystemJ statements continued

(a) Inductive definition of Term

(b) Inductive definition of Move

first clause checks for the presence of the signal \( \sigma \) while the other checks for its absence \( \neg \sigma \). The \( \text{Inst} \) rule does not check \( \sigma \) for the \text{abort} and \text{suspend} statements since they are non-immediate. \( \text{Inst} \) of sequential \((s_1; s_2)\) and synchronous parallel statements \((s_1||s_2)\) checks whether both \(s_1\) and \(s_2\) can be instantaneously terminated i.e. contain no \text{pause} statements. Hence, they are both expanded to the logical conjunction of \( \text{Inst} \) of \(s_1\) and \(s_2\), respectively. Finally, \( \text{Inst} \) of \text{while} is evaluated to \( \text{Inst}(s) \). Note that since the \text{while} block should contain at least one \text{pause} (Section 2.2), \( \text{Inst} \)(\text{while(true)} \(s) \) should never evaluate to \( \top \) for any correct SC-SystemJ program.

\( \text{Enter} \) condition of SC-SystemJ statements are shown in Figure 3.4b. Control-flow cannot \text{enter} (i.e. consume a logical tick) the signal and channel declarations and \text{emit}
3.5 Transition semantics of SC-SystemJ programs

statements. Hence, \textit{Enter} of these statements are evaluated to \(\perp\). On the other hand, \textit{Enter}(l : \texttt{pause}) results in \(Xl\), which means the clock-domain will settle at state labelled \(l\) at the end of the current program transition. Inductive definitions of \textit{Enter} for \texttt{abort}, \texttt{suspend} and \texttt{while}, are simply \textit{Enter} of their enclosing bodies. Similar to \textit{Inst}, \textit{Enter} of \texttt{present} is expanded to the disjunction of two conjunctive clauses and their evaluations are dependent on the status of the signal \(\sigma\). For the sequential statement \(s1; s2\), the inductive rule has two clauses which check whether control-flow can settle on \(s1\), or instantaneously terminate \(s1\) and settle on \(s2\). Finally, \textit{Enter} of the parallel statement \(s1 || s2\) has three clauses; whether control-flow can settle at either one of them, or on both simultaneously.

The rest of the definitions are shown in Figure 3.5, where \textit{Term} and \textit{Move} are defined in Figure 3.5a and Figure 3.5b, respectively. By definition (see Equation 3.3), \textit{Term}(s) requires that program control-flow to have previously settled on \(s\) and now terminate (exit) \(s\). Hence, \textit{Term} of all instantaneous statements are evaluated to \(\perp\). One can then easily see that \textit{Term}(l : \texttt{pause}) := l. \textit{Term} of \texttt{present} checks whether the control-flow can terminate the \texttt{present} \((s1)\) or \texttt{else} \((s2)\) branch similar to \textit{Inst} and \textit{Enter}. Note that signal checks for \(\sigma\) are not necessary in these clauses since the control-flow is already in the body of the \texttt{present-else} statement. On the other hand, \texttt{abort} can only be terminated when the control-flow is in the body of \texttt{abort} and the signal \(\sigma\) is present, or when it exits the \texttt{abort} body normally. For \texttt{suspend}, control-flow can only terminate the body when the signal is absent, i.e. \(\neg\sigma\) holds. \textit{Term} of sequential and parallel statements are defined similar to \textit{Enter}. \textit{Term} rules check for every possible combination of terminating conditions for these statements. Lastly, the \texttt{while} loop can never be terminated, hence it is evaluated to \(\perp\).

Program \(s\) can \textit{Move} from one state to another if there are at least two \texttt{pauses} in \(s\) or a single \texttt{pause} in a loop. Therefore, \textit{Move} of all statements that violate these conditions are evaluated to \(\perp\). \textit{Move} of \texttt{present} is similar to \textit{Term} except that there are additional next time conditions (\(\neg Xin(s2)\) and \(\neg Xin(s1)\)) that make sure the control-flow does not jump to the other branches. Note that \texttt{abort} can only \textit{Move} when the signal \(\sigma\) is absent. For \texttt{suspend}, control-flow \textit{stutters} when \(\sigma = \top\) and control-flow is already \textit{inside} the body of \texttt{suspend}. Control-flow can move inside \texttt{suspend} when \(\sigma = \perp\). \textit{Move} of the sequential statement \(s1; s2\) is defined as three disjunctive clauses: (1) the program control-flow can move inside \(s1\) while \texttt{not} in \(s2\), (2) vice-versa, and (3) it terminates \(s1\) and enters \(s2\). There are five conditions for which control-flow can move inside the parallel statement \(s1 || s2\): (1) when it is currently only in \(s1\), (2) only in \(s2\), (3) in both \(s1\) and \(s2\) simultaneously, (4) in both \(s1\) and \(s2\) and \(s1\) terminates, and (5) in both \(s1\) and \(s2\) and \(s2\) terminates. Lastly for the \texttt{while} statement, the control-flow either can move inside the loop body or terminate the body and re-enter the loop, i.e. \textit{Term}(s) \(\land\) \textit{Enter}(s).

These inductive rules are applied to the SC-SystemJ program until the DNF ofEqua-
Semantics of the SC-SystemJ Language

\[ \text{Inst}(\text{input}|\text{output} \text{ signal } \sigma \text{ op } z = \text{number}) := X\gamma_1 \]
\[ \text{Inst}(\#\sigma = \text{dataexpr}) := X\gamma_2 \]
\[ \text{Inst}(\text{if}(\text{expr}) s_1 \text{ else } s_2) := \gamma_3 \land \text{Inst}(s_1) \lor \neg \gamma_3 \land \text{Inst}(s_2) \]
\[ \text{Inst}(\text{extern symbol}(\text{dataexpr}_1, \ldots, \text{dataexpr}_n)) := X\gamma_4 \]

(a) Definitions of \text{Inst} for data statements

\[
\begin{align*}
  f : \gamma_1 & \to [\text{number}]_{\sigma} \\
  f : \gamma_2 & \to [z_{\sigma}(\text{dataexpr})]_{\sigma} \\
  f : \gamma_3 & \to [\text{expr}]_{\sigma} \\
  f : \gamma_4 & \to \lambda(\text{dataexpr}_1, \ldots, \text{dataexpr}_n)
\end{align*}
\]

(b) Mapping of propositions \gamma_n

Figure 3.6: Definitions for SC-SystemJ data statements

Figure 3.3 consists of only atomic propositions such as program labels and signals.

Data-flow semantics

Definitions for the SC-SystemJ data statements are shown in Figure 3.6. Note that only \text{Inst} of these statements are shown in Figure 3.6a since none of their executions consume a tick, i.e., they are instantaneous statements. Hence, \textit{Move}, \textit{Enter}, and \textit{Term} of these statements are all \(\perp\). Programmers specify the associative and commutative operator \(z\) in the signal declaration statement for combining multiple emissions of the same valued signal. This operator combines multiple values, which are emitted multiple times in a single program transition, into a single value. The signal value is assigned to an evaluated result of an expression using the hash operator \(#\). Conditional data expressions can be checked using the \textit{if-else} statement. Lastly, an external function can be called using the \textit{extern} function with optional arguments \text{dataexpr}_n separated by commas.

\text{Inst} of every data-flow statement results in a unique proposition \(\gamma\) in the equivalent LTL formula. For instance, all of the data statements except \textit{if-else} are defined as \(X\gamma\), which means that \(\gamma\) holds in the next state. Indeed, similar to \text{Inst}(\text{emit } \gamma)\) shown in Figure 3.4a, the result of data computations is also delayed by one tick. \text{Inst} of \textit{if-else} in the data-flow semantics is akin to \textit{present-else} statement in Figure 3.4a except that \text{Inst} of \(s_1\) and \(s_2\) are logically connected with \(\gamma\) instead of the signal proposition \(\sigma\). Let \(D\) be a set of data computations in the SC-SystemJ program, then a function \(f : \gamma \to d\), such that \(d \in D\), maps a proposition \(\gamma\) to a valuation \(d\) as shown in Figure 3.6b. Here, a valuation of some expression is denoted by \([expr]_{\sigma}\) whose result is assigned to a signal \(\sigma\), whereas \([expr]\) returns a result without assignment. For the mapping of \(\gamma_2\), operator \texttt{op } z_{\sigma} : v \to \mathbb{Z}_{\geq 0}, z_{\sigma} \in \{+,*\} \) is defined as \text{dataexpr} + c_{\sigma} for \(d = +\) and \text{dataexpr} \ast c_{\sigma} for \(d = \times\).
3.5 Transition semantics of SC-SystemJ programs

for $d = \ast$, where $c_\sigma$ is the current value of a signal $\sigma$. Finally, the mapping of $\gamma_4$ returns a function $\lambda$ whose mapping is dependent on the host language’s function (C or Java etc.). These inductive rules build the transition system in Equation 3.3 or equivalently the system defined in Definition 1. Given that the transition system for a clock-domain is described, we are now ready to define properties of the SC-SystemJ clock-domain; primarily determinism and reactivity.

Reactivity and determinism of a SC-SystemJ clock-domain

The following definitions are applied to a single SC-SystemJ clock-domain, which is a pure synchronous program.

**Definition 6. Reactivity:** A transition system $M$ is reactive if in any of its state $q$ and for any monomial $b$ involving its input symbols, there is a transition out of $q$ with $b'$ as the guard such that $b$ satisfies guard $b'$.

Consider two simple transition systems as shown in Figure 3.7. Each is composed of two states $S1$ and $S2$, and two inputs $a$ and $b$. In this example, both output signals and actions for all transitions are $\emptyset$, and omitted for simplicity. The system shown in Figure 3.7a is reactive because it can take a transition out of both states with every combination of input signals, i.e. $a \land b$, $\neg a \land b$, $a \land \neg b$ and $\neg a \land \neg b$. On the other hand, the system in Figure 3.7b is non-reactive since there are certain combinations of the input signals, e.g. $\neg a \land \neg b$, which do not have a corresponding outgoing transition from the states $S1$ or $S2$.

![Figure 3.7: Reactivity of the SC-SystemJ transition systems](image)

**Definition 7. Determinism:** A transition system $M$ is deterministic if in any of its states $q$, if there are two outgoing distinct transitions $(q, b_1, O_1, Y_1, A_1, q_1)$ and $(q, b_2, O_2, Y_2, A_2, q_2)$, then $b_1 \land b_2$ is false.

Consider another pair of transition systems shown in Figure 3.8. Both transition systems can take a transition to either state $S2$ or state $S3$ from state $S1$ depending on
an input symbol $a$. Similar to Figure 3.7, output signals and actions are omitted in this figure, since they are all $\emptyset$. In Figure 3.8a, there are two distinct transitions from $S_1$ with the guards $a$ and $\neg a$, respectively. Since there is only one unique input symbol for the transition to $S_2$ and to $S_3$, the system is deterministic. On the other hand, Figure 3.8b is non-deterministic, since the system can make a transition to either $S_2$ and $S_3$ for the same input $a$.

![Deterministic system](image1)

![Non-deterministic system](image2)

Figure 3.8: Determinism of the SC-SystemJ transition systems

### 3.5.2 Transition semantics of a network of clock-domains

Given two transition systems, $M = (Q_1, st_1, I_1, O_1, Y_1, A_1, T_1)$ and $N = (Q_2, st_2, I_2, O_2, Y_2, A_2, T_2)$, then the asynchronous composition is defined as follows: $M \bowtie N = (Q_1 \times Q_2, (st_1, st_2), I_1 \cup I_2, O_1 \cup O_2, Y_1 \cup Y_2, A_1 \cup A_2, T)$, where

$$T := \begin{pmatrix}
(q_1, q_2), b_1, O'_1 \cup O'_2, (q'_1, q'_2))|\in T_1 \\
(q_1, q_2), b_2, O'_2 \cup O'_1, (q'_1, q'_2))|\in T_2 \\
(q_1, q_2), b_1 \land b_2, (O'_1 \cup O'_2), (Y'_1 \cup Y'_2), (A'_1 \cup A'_2), (q'_1, q'_2))|\in T_i, i = 1, 2
\end{pmatrix}
$$

Note, that Equation 3.4 is *different* from the asynchronous product defined in CCS [59], which consists of the so called rendezvous transition (simultaneous transition of both the automata with shared communication actions). The handshake mechanism in the SC-SystemJ model is implemented using input and output signals and hence does not have a rendezvous transition. The implementation of SC-SystemJ channel is described in the following section.
Channel rewrites

The channel implementation in SC-SystemJ does not halt the complete CD, which differs from CCS/CSP [59, 26] and CRSM [23]. Instead, the SC-SystemJ channel is implemented using a four phase handshake mechanism via signals, shown in Figure 3.9. In Figure 3.9, two clock-domains, called ‘Sender’ and ‘Receiver’ implement channel communication using signals called ACK and REQ. For each channel declaration there is also a corresponding pair of ACK and REQ signals declared. These signals are hidden to SC-SystemJ programmers in order to prevent their intervention in channel communications between clock-domains. The channel send and receive statements are rewritten into a sequence of SC-SystemJ kernel statements as shown in Algorithm 3.1 and Algorithm 3.2, respectively – note that the presented handshake mechanism, in these algorithms, leads to blocking at the logical tick level rather than halting the CD.

The main reason for rewriting the channel statements is to preserve reactivity [10] and scalability of the SC-SystemJ program. A reactive system by definition, is a system which has at least a single state transition for every given input signal that results in
the production of a set of output signals [10]. Therefore, unlike CSP, a SC-SystemJ CD blocks only at the logical tick on send and receive statements and allows other reactions to capture inputs, perform internal transitions, and produce output signals. Ramesh [60] has shown the impossibility of robust channel communication when preemption is involved during rendezvous (e.g. via \textit{abort} construct in SC-SystemJ) in CRP [22] programs. This is due to timing issues in the communication between the main CRP processes and a special process, called \textit{coordinator}, which runs alongside the main processes and handles all rendezvous communications. The problem is addressed by relaxing the reactivity of the program, which discards inputs from the environment at certain crucial times during rendezvous between communicating nodes. On the other hand, SC-SystemJ does not require an additional coordinator clock-domain for rendezvous communication.

In contrast to the previous approaches, SC-SystemJ CDs are reactive even though preemption may occur during the rendezvous. Consequently, there are three possible scenarios that might arise from the SC-SystemJ channel implementation, shown in Figure 3.10:

**Scenario-1** Both CDs complete rendezvous without any preemption. This is the case when the program is written such that there are no statements nor input events from the environment that can preempt the rendezvous operation between reactions in different CDs. Figure 3.10a illustrates such a scenario. Consider the black and grey circles which show the execution traces of the sender and the receiver CDs, respectively. The sender first enters a busy wait state, waiting for the signal ACK coming from the receiver (as shown in Algorithm 3.1 lines 1-3). Unlike the sender, the receiver can preempt the busy wait state since it checks for the absence of the signal REQ. The receiver then continuously emits the acknowledgement signal (ACK) until it receives the signal REQ from the sender (Algorithm 3.2 lines 4-7).
3.5 Transition semantics of SC-SystemJ programs

The signal ACK preempts the sender and it enters a state that continuously emits REQ. Finally, the receiver completes the rendezvous first, followed by the sender (denoted by done in Figure 3.10a).

**Scenario-2** One CD preempts during rendezvous and re-enters the rendezvous state. It is clear from Algorithms 3.1 and 3.2 that CDs will eventually finish rendezvous provided that there is no execution path in the program such that the preempted reaction *never* reenters the same rendezvous point. Figure 3.10b shows this case where the sender CD preempts before the receiver catches the REQ signal. CDs then synchronize later when the sender re-enters the rendezvous state.

**Scenario-3** One CD preempts during rendezvous and never re-enters the rendezvous state. This can result in deadlock where the sender or receiver never completes the rendezvous. Figure 3.10c illustrates this scenario where the sender preempts while it is in the REQ state and the receiver remains forever blocked in the ACK state.

Deadlock is possible for **Scenario-3** since preemption can arbitrarily change the execution order of channel statements. For instance, consider the example program in Figure 3.11.
where the signal \( O \) will \textit{almost always} be emitted iff the signal \( A \) is always absent. Consider the scenario wherein signal \( A \) preempts the second CD while both CDs are in the rendezvous state of the channel \( C_2 \). Consequently, both CDs are now at different rendezvous points such that CD2 will try to rendezvous with CD1 on \( C_1 \) due to the enclosing loop, while CD1 is still blocked, waiting to complete a rendezvous on \( C_2 \). Therefore, CD1 and CD2 will never be synchronised resulting in a deadlock.

Such deadlock scenarios can be detected through functional verification, specifically using safety properties, which will be presented in the next chapter. Alternatively, one can use a programming style to write all rendezvous statements in separate synchronous parallel reactions without any preemting statements enclosing them.

\textbf{Reactivity and determinism of a network of clock-domains}

An SC-SystemJ GALS program consists of a network of clock-domains which run asynchronously and communicate with each other when necessary, in order to achieve a common goal. In this section, a sketch of a proof for the reactivity of a network of clock-domains is given. The asynchronous composition of reactive clock-domains may introduce non-deterministic behaviour, which is also discussed next.

\textbf{Theorem 3.5.1.} Given two transition systems \( M \) and \( N \) representing reactive clock-domains \( CD_1 \) and \( CD_2 \), respectively, then the asynchronous composition of \( CD_1 \) and \( CD_2 \) (Equation 3.4) resulting in the transition system \( K \), is also reactive.

\textit{Proof.} Equation 3.4 states that the transition system resulting from the composition of two asynchronous clock-domains can have one of three possible state transitions: (1) transition of the first clock-domain, (2) transition of the second clock-domain, and (3) transition of both clock-domains. Furthermore, Definition 6 states that there is always a transition out of any of the states in \( M \) or \( N \) for any combination of input events. Then for any state in the transition system \( K \), there is also at least one outgoing transition for any given input event due to the third clause in Equation 3.4. Therefore, given that the clock-domains for the transition systems \( M \) and \( N \) run infinitely fast such that no input events can ever be missed from their respective environments, transition system \( K \) is also reactive.

\( \Box \)

On the other hand, given two clock-domains, which are deterministic, their asynchronous composition might be non-deterministic. To illustrate this, consider the example shown in Figure 3.12. Two transition systems representing clock-domains named \( CD_1 \) and \( CD_2 \) are shown in Figure 3.12a. Each clock-domain consists of three states. States \( A \) and \( D \) are the initial states of \( CD_1 \) and \( CD_2 \), respectively. Each clock-domain makes only a single transition from the initial state and settles on the resultant state forever.
According to Definition 7, each of these clock-domains is deterministic and reactive; there are two distinct transitions out of the initial states, which are guarded by input symbols $a$, $\neg a$ and $b$, $\neg b$, respectively. On the other hand, when these clock-domains are combined using the asynchronous composition given in Equation 3.4, the resultant transition system is non-deterministic as shown in Figure 3.12b. For example, for an input symbol $a$, the transition system in Figure 3.12b can possibly make a transition to one of three states from its initial state $(A, D)$: (1) to $(B, D)$ when only $CD1$ makes a transition or (2) to $(A, F)$ when only $CD2$ makes a transition or (3) to $(B, F)$ when both $CD1$ and $CD2$ make transitions. This is due to the fact that clock-domains can interleave in the asynchronous composition (i.e. a clock-domain can make a state transition while others do not). In other words, state transitions of one clock-domain do not depend on others.

Non-deterministic nature of GALS programs is also well illustrated by the example shown in Figure 3.13. Here, CD1 tries to synchronise with CD2 in a loop (line 3) via a

```plaintext
{ /* CD-1 */
  output channel C;
  while(true){send C;}
}

{ /* CD-2 */
  input channel C;
  input signal A;
  output signal O1;
  output signal O2;
  {while(true){ receive C; emit O1; }}
  {while(true){ await(A); emit O2; }}
}
```

Figure 3.12: Non-deterministic GALS program
channel named C. CD2 blocks until the input signal A becomes present, which then emits O2 (line 13). However, when the execution of receive completes (line 11), CD2 can also emit O1. In order to ensure determinism, the observed behaviour of the system should be deterministic [61], i.e. for the same sequence of inputs the system should always generate the same sequence of outputs. Under this assumption, the program in Figure 3.13 is non-deterministic since for the same input A the system generates two different outputs; sometimes O1 only, and sometimes both O1 and O2. On the other hand, it should be noted that an individual clock-domain, which follows synchronous MoC, in Figure 3.13 is deterministic.
Compilation and Verification of SC-SystemJ Programs

The previous chapter presented inductive definitions of SC-SystemJ statements which can be used to translate a whole SC-SystemJ program into a set of propositional LTL formulas, one for each clock-domain. This chapter will introduce how these formulas can be converted into a network of Mealy automata using a novel and efficient compilation algorithm. The resultant network of Mealy automata have a straightforward syntactic translation into Promela code for verification via the SPIN model-checker or can be translated into 'C/Java' code for execution on a target platform.

Significant amount of research literature targets verification of programs designed in synchronous languages like Esterel. Halbwachs et al. present an approach using observer-based verification for synchronous programs. However, they only concentrate on safety properties – verifying that something bad never happens. Observer based verification of imperative Esterel programs is also carried out in, but unlike Halbwachs et al., the authors in automatically generate observers in Esterel from LTL properties. The advantage of using observer-based verification over LTL properties is flexibility – the user can program complex observers in the programming language itself and safety property verification using observers can be translated into reachability analysis. The earlier versions of Esterel compilers, such as v2 and v3, also generate an automaton for an entire Esterel program. However, both these versions of the compilers have been aban-
doned because they carried out a syntactic translation of the source code using textual (or bit-state) regular expression matching, this lead to a large explosion in state space [64, 65], thereby making the compiler run out of memory during compilation. The circuit (netlist) based compilers [56] designed to overcome the non scalable nature of automaton Esterel compilers [10], convert netlist descriptions of Esterel programs into FSMs. These FSMs are then reduced using bisimulation equivalence and verified using observers. In contrast, our approach performs semantic analysis for building the automaton, and produces Promela processes from the SC-SystemJ program for verification based on using the SPIN model-checker. Moreover, it supports both safety and liveness (i.e. something good eventually happens) property verification of GALS programs developed in SC-SystemJ.

The approach for automatically generating Promela code for verification of SC-SystemJ programs is not new. Ramesh et al. [66] developed a toolset for converting Communicating Reactive State Machines (CRSM) to Promela code for verification of GALS programs. Although similar to our approach on a cursory look, their approach differs from ours in the way rendezvous communication between synchronous program islands interplays with reactivity. Whereas our approach can be extended to real-time verification, their approach can not. This is mainly due to their approach of giving up on reactions to input events at crucial moments during the rendezvous. A similar problem is inherited by Communicating Reactive Processes (CRP) [67], which is another GALS extension of Esterel. A number of other attempts have been made previously to encapsulate synchronous programs into asynchronous models. For example, [68, 69, 70] all verify GALS systems by performing a shallow embedding of the compiled synchronous programs into Promela or some other model-checking engine designed to verify asynchronous systems. In all these cases the communication between synchronous program islands is not programmed by the system designer, but rather added to the program at the verification stage using some specific features of the model-checker at hand. This breaks the What You Prove Is What You Execute (WYPIWYE) philosophy, which is the goal of the presented work.

Section 4.1 gives an overview of the tool-chain flow for the compilation and verification of the SC-SystemJ program. In particular, generation of Mealy automaton for synchronous and GALS programs, and corresponding C codes, is presented in Sections 4.1.1 and 4.1.2 respectively. Section 4.1.3 discusses how classical issues of reactive languages are addressed in the new compilation strategy. Section 4.2 describes the verification strategy for SC-SystemJ programs and presents benchmark results for the ice cream manufacturing facility case study. Finally, Section 4.3 gives the additional experimental results.
4.1 Correct by construction code generation

The main advantage of the compilation techniques presented in this chapter is the ability to verify properties of programs before deployment. The technique presented in this chapter transforms SC-SystemJ program into an intermediate format, which is a network of FSMs. Promela code is generated from this intermediate format. These Promela models, which capture the behaviour of the SC-SystemJ program, are used for verifying the functional correctness properties via the SPIN model-checker \[44\]. The verified model is then used for carrying out a simple syntactic translation into 'C/Java' executable, thereby guaranteeing the WYPIWYE paradigm as stated by Berry [35]. The WYPIWYE paradigm has been successfully applied to compilation of existing synchronous languages with a formal semantics such as Esterel [10], Lustre [12], and Signal [11]. Model-checking programs developed in synchronous languages have been successful since these languages are based on a formal semantics. The semantics for both sequential and concurrent computation allows one to generate models or embed the compiler generated code into model-checkers, which can then be used to verify properties of programs. Model-checking, with its complete state space exploration and counter example (witness) generation, greatly complements the standard programming approach of unit testing for bug detection in safety-critical software programs. Automatic verification and realisation of a design at system-level also has been an increasing demand in many system-on-chips (SoC) design approaches. Authors in [71] developed a correct-by-construction technique to enable automatic verification/generation of interfaces between IPs to reduce design effort on verifying functionality and timing of SoC designs. Their work extends the structured methodology called the reuse methodology manual (RMM) [72], which is SoC design flow adopted by industry, that incorporates formal verification for ensuring correctness of an overall system. In this thesis, the WYPIWYE approach is extended to safety-critical programs developed in GALS languages, in particular SC-SystemJ.

The proposed WYPIWYE tool-chain flow for generating WYPIWYE executables is shown in Figure 4.1. The tool-chain flow is as follows:

1. An informal (natural language) control logic and (optional) plant model descriptions are converted into SC-SystemJ programs.

Here, control logic is a program which can be distributed across hardware control devices to control operations of the system. On the other hand, the plant is the environment, providing input stimuli to this control system. Control logic and the plant model are then combined together to form a single SC-SystemJ program, making the overall system a closed loop control system for verification. This technique is required for verifying certain types of functional properties of a system that make
certain assumptions on the environment, e.g. the system will halt when the heater or air conditioning is not working. Alternatively, the system can be left open, i.e. without a plant model, for verifying the properties which should hold regardless of the event generated from the environment, e.g., verifying the absence of deadlocks in the control logic.

2. Every clock-domain (controller and possibly combined with plant) is compiled using the inductive semantic transition rules presented in Chapter 3.

Every SC-SystemJ clock-domain is first converted into a propositional LTL formula. These formulas are then converted into a network of Labelled Generalized Büchi Automata (LGBA), a type of finite state machine that accepts an infinite input sequence. Finally, a novel algorithm (cf. Algorithm 4.1) is applied to transform these automata into a network of deterministic Mealy automata.

3. Generate a network of Promela processes and verify properties on the program

The network of Mealy automata generated in Step-2 is then converted into a network of Promela processes which are given as the input to the SPIN model-checker. The
equivalence between the SC-SystemJ model and the corresponding Promela model is explained in Section 4.2.1. Properties, specified as LTL formulas are checked on these Promela processes.

4. Generate an executable

If all specified properties are verified with SPIN in Step-3, then the SC-SystemJ control logic is directly compiled into executable C or Java code, using a straightforward syntactic translation technique, and finally deployed on the target execution platform. If the verification stage fails, a witness (fault trace) is generated which can be used to update the control logic or original specification of the system.

The compilation of SC-SystemJ programs is non-trivial due to a number of issues such as: instantaneous loops, synchronous (i.e. due to signals) and asynchronous deadlocks (i.e. due to channels). In this thesis, these problems are addressed using the model-checking approach rather than introducing new compilation techniques for two main reasons: (1) verifying properties on programs inherently guarantees freedom from the aforementioned issues and (2) asynchrony in SC-SystemJ programs leads to an exponential increase in state space, and model-checkers already curtail this growth in state space using techniques such as partial order reduction, thereby reinventing and re-implementing these techniques in the compiler is an exercise in futility.

In the following sections, the compilation technique for generating a network of Mealy automata is presented. This Mealy automata is an intermediate format used for generating Promela code, for verifying properties of the SC-SystemJ program, or for generating executable code (C/Java) of the verified program.

4.1.1 Generating a Mealy automaton for a synchronous clock-domain

The generation of the Mealy automaton for an individual CD is carried out in two steps. In the first step, a Labelled Generalized Büchi automaton (LGBA) is created using the tableau approach described by Gerth et al. [73] from the LTL formula describing the SC-SystemJ clock-domain (Figure 4.1). LGBA is used as the input source to build a deterministic Mealy automaton, which is a state transition system for the SC-SystemJ clock-domain. The tableau algorithm to generate the LGBA is not given here since it is not a contribution of this thesis, and readers are referred to [73] for the complete algorithm.

In the second step, a novel algorithm is used to transform the LGBA into a Mealy automaton representing the state transition system for a single clock-domain. The background required to understand the LGBA created from the LTL formula is given next:
Definition 8. **Generalized Büchi Automaton** (GBA) is a quadruple \((Q, I, \rightarrow, F)\), where \(Q\) is the finite set of states, \(I \subseteq Q\) are the initial states, \(\rightarrow \subseteq Q \times Q\) is the transition relation and \(F \subseteq 2^Q\) is a set of sets of accepting states, where \(F\) may be empty.

GBA accepts an infinite input sequence \(\sigma = q_0, q_1 \ldots\) such that, \(q_0 \in I\) and, for each \(i \geq 0, q_i \rightarrow q_{i+1}\). An accepting execution \(\sigma\) is an execution such that, for each acceptance set \(F_i \in F\), there exists at least one state \(q \in F_i\) that appears infinitely often. Figure 4.2 shows an example of GBA with examples of three runs \(\sigma_1, \sigma_2\) and \(\sigma_3\). Note that double circled state \(S_1\) denotes an accepting state. Here, \(\sigma_1\) and \(\sigma_2\) are accepted execution traces since they both visit \(S_1\) infinitely often. On the other hand \(\sigma_3\) is rejected, because \(S_1\) appears only once in the execution trace.

Definition 9. A **Labelled Generalized Büchi Automaton** (LGBA) is a triple \((A, D, L)\), where \(A\) is the GBA, \(D\) is the domain of propositions and \(L\) is the labelling function: \(L : Q \rightarrow 2^D\).

LGBA is another variant of Büchi automaton whose input is associated with labels instead of states. Formally, a LGBA accepts a word \(\zeta = x_0, x_1 \ldots\) from \(D^\omega\) iff there exists an accepting execution \(\sigma = q_0, q_1, \ldots\) of \(A\), such that for each \(i \geq 0, x_i \in L(q_i)\).

From [73], it is known that the generated LGBA using the tableau method only accepts the language (i.e. \(\zeta\)) that satisfies the LTL formula. In order to demonstrate the conversion from the LTL formula representing the SC-SystemJ clock-domain to a LGBA, the transition system for a simple SC-SystemJ single clock-domain program in Figure 4.3a is shown in Figure 4.3b, where the program locations are specified using labels \(L_1, L_3\), and \(L_4\). Such a symbolic transition system is built from the SC-SystemJ clock-domain using the transition relation given in Equation 3.3, where each clause represents instantaneous, enter, terminate and move transition of the SC-SystemJ clock-domain, respectively. The first (Inst) clause of this formula indicates that no instantaneous transition can be taken from the starting state \(st\), since the program consists of pause statements. Therefore, the first clause in Figure 4.3b is \(\bot\). The second (Enter) clause states that from the starting

![Diagram of GBA](image-url)
4.1 Correct by construction code generation

{  
    input signal A;  
    output signal B;  
    present(A){  
        emit B;  
        L1: pause;  
    }  
}  
||  
{  
    L3: pause;  
    L4: pause;  
}  

(a) Simple SC-SystemJ program

(b) Generated transition system

Figure 4.3: Example transition system generated from the SC-SystemJ semantics

state \( st \) the next transition will be to states \( L1 \land L3 \) with emission of signal \( B \) if signal \( A \) is present. Otherwise, the program will only make a transition to the state \( L3 \). The third (\( Term \)) clause states that the program terminates if it makes a transition away from all of the states: \( L1, L3 \) and \( L4 \). Finally, the program will make a move from either one of \( L3 \land L1 \) or \( L3 \) to \( L4 \) before terminating (clause 4). This behaviour is clearly evident from the program in Figure 4.3a.

A derivation tree for the \( Inst \) clause for the program in Figure 4.3a is shown in Figure 4.4 which is read in a bottom-up fashion. Given the program \( p \) in Figure 4.3a, the control-flow first encounters the synchronous parallel operator || with reactions \( s1 \) and \( s2 \), which is rewritten to a logical conjunction as given by the inductive rule for \( Inst \) (Figure 3.4a). Assume that the inductive rule is first applied to the reaction \( s1 \) (the order of application is inconsequential) in which case, two signal declaration statements for \( A \) and \( B \) are performed sequentially before the \( present \) statement. Declaring signals is instantaneous, i.e. it does not consume a tick, hence, these statements both result in \( \top \). \( Inst(present) \) in the reaction \( s1 \) results in \( A \land XB \land \bot \lor \neg A \), see the inductive rule in Figure 3.4a. The result of the derivation for \( s1 \) is then \( A \land XB \land \bot \lor \neg A \). Next, since \( Inst(pause) \) is \( \bot \), one can easily see that \( Inst(s2) \) is also \( \bot \). Finally, \( \bot \) is connected to the rest of the formula with a conjunction therefore, the overall result of the derivation for \( Inst(s1)||s2) \) is also \( \bot \).

A derivation tree for the second clause of Figure 4.3b, i.e. \( Enter \), is shown in Figure 4.5. Here, \( Enter \) of two parallel reactions, \( s1 \) and \( s2 \), is further decomposed into three clauses connected with logical disjunction (Figure 3.4b). However, since it is already known that \( Inst \) of the reaction \( s2 \) is \( \bot \), the only interesting clauses are the first and the last one, which are \( Enter(s2) \land Inst(s1) \land X\neg \text{in}(s1) \) and \( Enter(s1) \land Enter(s2) \), respectively. The
\[
\frac{
\bot \\
\top \wedge (A \land XB \land \bot \lor \neg A) \land Inst(L3 : \text{pause}; L4 : \text{pause})
}{
\top \wedge \{ \text{present}(A) \{ \text{emit } B; L1 : \text{pause} \} \} \wedge Inst(s2)
}
\]

\[
\frac{
\top \wedge \{ \text{output signal } B \} \wedge Inst(s1) \wedge Inst(s2)
}{
\text{Inst(input signal } A \} \wedge Inst(s1) \wedge Inst(s2) 
}
\]

\[
\frac{
\text{Inst}(s1) \land \text{Inst}(s2)
}{
\text{Inst}(s1 \| s2) \land \text{Inst}(p)
}
\]

Figure 4.4: Derivation tree deducing \text{Inst} for SC-SystemJ program in Figure 4.3a

\[
\frac{
\text{st} \land \neg XL4 \land XL3 \land \neg A \land \neg XL1 \lor
\text{st} \land A \land XL1 \land XB \land \neg XL4 \land XL3
}{
\text{st} \land \top \land \text{Enter}(L3 : \text{pause}; L4 : \text{pause}) \land \neg A \land \neg XL1 \lor

\frac{
\text{st} \land \text{Enter}(s2) \land \text{Inst}(s1) \land \neg Xin(s1) \lor
\text{st} \land \text{Enter}(s1) \land \text{Enter}(s2)
}{
\text{st} \land \text{Enter}(s1 \| s2) \land \text{Enter}(p)
}
\]

Figure 4.5: Derivation tree deducing \text{Enter} for SC-SystemJ program in Figure 4.3a

\[
\neg \text{st} \land \neg XL1 \land \neg XL3 \land \neg XL4
\]

Figure 4.6: Derivation tree deducing \text{Term} for SC-SystemJ program in Figure 4.3a

Induction rule for \text{Enter}(s1) gives \top until it reaches the \text{present} statement. According to Figure 3.4b, \text{Enter}(\text{present}(A) \{ \text{emit } B; L1 : \text{pause} \}) gives \top \land XL1 \land XB. Similarly, it can be easily seen that \text{Enter}(s2) is \neg XL4 \land XL3. It is already known, from the previous derivation for \text{Inst} (Figure 4.4), that \text{Inst}(s1) is \neg A. Finally, \text{in}(p) collects all labels inside the program body \text{p}, see Equation 3.1, thus, \neg Xin(s1) is rewritten to \neg XL1.

Deriving the \text{Term} transition is quite trivial, which is shown in Figure 4.6. \text{in}(p) simply collects all labels inside the program body \text{p}. The resultant formula is a logical conjunction of these labels.

Derivation of the \text{Move} transition is given in Figure 4.7. Since \text{Move}(s1) results in \bot \land Move(s1 \| s2) is simplified to two clauses, which are shown as (1) in Figure 4.7. The inductive definition for \text{Move}(s2) further requires rewriting \text{Move}(L1 : \text{pause}; L2 : \text{pause}) whose result is shown as (2). Finally \text{Term}(s1), results in \bot, and after applying Equation 3.1 to the remaining control predicates (i.e. \text{in}) the final formula (3) is shown in
4.1 Correct by construction code generation

\[
\begin{align*}
\left( L3 \land \neg X L3 \land \neg L4 \land X L4 \land \neg L1 \land \neg X L1 \lor L3 \land \neg X L3 \land \neg L4 \land X L4 \lor L1 \land \neg X L1 \right) \\
\left( L3 \land \neg X L3 \land \neg L4 \land X L4 \land \neg in(s1) \land \neg X in(s1) \lor L3 \land \neg X L3 \land \neg L4 \land X L4 \land Term(s1) \land \neg X in(s1) \right)
\end{align*}
\]

\[
\begin{align*}
\left( \text{Move}(s2) \land \neg in(s1) \land \neg X in(s1) \lor \text{Move}(s2) \land Term(s1) \land \neg X in(s1) \right) \\
\text{Move}(s1|s2) \\
\text{Move}(p)
\end{align*}
\]

Figure 4.7: Derivation tree deducing \textit{Move} for SC-SystemJ program in Figure 4.3a

Figure 4.7 shows the LGBA generated by the tableau approach of [73], which accepts only the words that satisfy the transition relation in Figure 4.3b. The propositional formula satisfied in each state is shown in the table in Figure 4.8. These states are labelled from the set \(2^D\) on the binary domain of propositions (including inputs and guarded actions). Note that the labelling function \(L\) is more in the spirit of satisfiability solving rather than the one given by [73], wherein the states are labelled with all subsets of \(2^D\) excluding the negative propositions satisfied by that state.

Consider the LTL formula in Figure 4.3b and its non-deterministic LGBA in Figure 4.8. The \texttt{false} (\(\bot\)) formula representing the \texttt{Inst}(p) transition is never accepted in the generated automaton. The \texttt{Enter}(p) (second clause in Figure 4.3b) is satisfied by the transitions from \texttt{N6} to \texttt{N9}. Similarly, the third (\texttt{Term}) and fourth (\texttt{Move}) clauses are satisfied by the transitions from \texttt{N5} to \texttt{N9} and the rest, respectively. Note that at this stage, all actions (e.g. signal emissions and data computations etc.) appear as propositional variables in the LGBA states. For example, the proposition \(P1\) in the state \texttt{N7} implies that “the
signal $B$ is being updated to $\top$. It is also interesting to note that the acceptance state \( \text{N9} \) has an empty set as its label and hence, all the accepted words are finite prefixes. Moreover, it is possible that unreachable states can be generated in the LTL formula and the corresponding LGBA depending on how the program is written. Note that all nodes that have an incoming guard from a fictional (or dummy) state called Init are initial states. For example, \text{N16}, \text{N15}, \ldots \text{N6} in Figure 4.8 are initial states. As it will be explained below, they are used internally by the compiler when generating a deterministic Mealy machine.

The LGBA is further processed by the compiler such that for each state in the LGBA, all the propositions other than the state labels are pushed onto the outgoing edges. For example, the outgoing edges of \text{N7} and \text{N6} are annotated with the propositions $P1$, $A$ and $\neg A$, respectively. These propositions are then removed from the propositional formulas in the original states. If the propositional formula in a state consists of only state labels, all its outgoing edges are annotated with $\top$ (e.g. outgoing edges of \text{N17}, \text{N16}, etc.). The result of this process is a graph, which is shown in Figure 4.9a.

In the next step, the generated graph is taken as an input, and then reduced to a deterministic Mealy automaton representing the transition system of the SC-SystemJ program. The pseudo-code for generating the Mealy automaton is shown in Algorithm 4.1. It has five main parts:

(a) The resultant graph after pushing non-state propositions to outgoing edges in the LGBA

(b) The resultant graph after merging equivalent states

(c) The resultant graph after removing Init

<table>
<thead>
<tr>
<th>Name</th>
<th>Proposition satisfied</th>
<th>Name</th>
<th>Proposition satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{N17}</td>
<td>$\neg L1 \land \neg L3 \land L4$</td>
<td>\text{N9}</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>\text{N16}</td>
<td>$L1 \land L3 \land \neg L4$</td>
<td>\text{N8}</td>
<td>$\neg L1 \land L3 \land \neg L4$</td>
</tr>
<tr>
<td>\text{N15}</td>
<td>$\neg L1 \land L3 \land \neg L4$</td>
<td>\text{N7}</td>
<td>$st$</td>
</tr>
<tr>
<td>\text{N13}</td>
<td>$\neg L1 \land \neg L3 \land \neg L4$</td>
<td>\text{N6}</td>
<td>$st$</td>
</tr>
<tr>
<td>\text{N11}</td>
<td>$L1 \land L3 \land \neg L4$</td>
<td>\text{N5}</td>
<td>$\neg st$</td>
</tr>
</tbody>
</table>

Figure 4.9: Creating a deterministic Mealy automaton
**Algorithm 4.1:** Pseudo-code for generating deterministic Mealy automaton from the LGBA

**Input:** labeled_graph_node set = \(
\{ \{ \text{name}: \text{string}; \text{incoming}: \text{string list}; \text{guards}: \text{set of formula}; \text{tlabels}: \text{formula} \} \ldots \} \)

**Output:** deterministic fsm

**Data:**
- node_tbl: Hashtbl<formula,labeled_graph_node>
- node_to_replace: Hashtbl<string,string>

/* Merging equivalent nodes */

1. foreach node ∈ labeled_graph_node set do
   2. if ∃ND ∈ node_tbl s.t., ND.tlabels = node.tlabels then
      3. Hashtbl.add node_to_replace node.name ND.name;
      4. ND.incoming ← ND.incoming ∪ node.incoming;
      5. ND.guards ← ND.guards ∪ node.guards;
   6. else
      7. Hashtbl.add node_tbl node.tlabels node;

**Data:**
- new_incoming: set of string

8. foreach node ∈ labeled_graph_node set do
   9. new_incoming ← ∅;
   10. foreach name ∈ node.incoming do
        11. if ∃ND.name ∈ node_to_replace with key name then
             12. new_incoming ← new_incoming ∪ {ND.name};
        13. else
             14. new_incoming ← new_incoming ∪ {name};
        15. node.incoming ← new_incoming;
   16. labeled_graph_node set ← Hashtbl.values node_tbl;

**Data:**
- guards_1: set of formula
- guards_2: set of formula
- init_nodes : labeled_graph_nodes set
- st_node : labeled_graph_node

/* Removing connections to the Init node */

17. foreach node ∈ labeled_graph_node set do
   18. guards_1 ← ∅;
   19. init_nodes ← ∅;
   20. if node.tlabels = “st” then
      21. st_node ← node;
   22. for c in 0 to |node.guards| do
        23. if node.incoming(c) ≠ “Init” then
             24. guards_1 ← guards_1 ∪ {node.guards(c)};
        25. else
             26. init_nodes ← init_nodes ∪ {node};
   27. /* Handling sub-formulas */
        28. node.guards ← \( \bigwedge_{g \in \text{guards}_1} \);  

**Data:**
- nodes_to_rep : set of labeled_graph_nodes
- nodes_to_replace ← \{init_nodes\} – \{st_node\};
- labeled_graph_node set ← labeled_graph_node\nodes_to_replace;

30. foreach node ∈ init_nodes do
   31. if ∃ND ∈ labeled_graph_node set s.t., node.tlabels is a sub-formula of ND then
      32. /* Similar to lines [1] to [27] */
   33. labeled_graph_node set ← perform_reachability_analysis(labeled_graph_node);
   34. deterministic fsm ← perform_optimisation(labeled_graph_node)
Algorithm 4.2: Reachability analysis

Function perform_reachability_analysis(labeled_graph_node):
    Data: N: a set of labeled_graph_node
    begin
    N ← ∅;
    foreach node ∈ labeled_graph_node do
        if reachable_st(node,labeled_graph_node) = ⊤ then
            N ← N ∪ {node}
    return N

Function reachable_st(node,labeled_graph_node):
    begin
    if node.incoming = ∅ ∧ node.name = st then
        return ⊤
    else if node.incoming = ∅ ∧ node.name ≠ st then
        return ⊥
    else
        foreach nname ∈ node.incoming do
            node ← get_parent_node(nname,labeled_graph_node);
            if reachable_st(node,labeled_graph_node) = ⊤ then
                return ⊤
        return ⊥
    end

Algorithm 4.3: Optimising the paths and extract actions from guards

Function perform_optimisation(labeled_graph_node):
    Data: fsm: a set of fsm_node
    Data: fsm_node: {name: string; incoming: string list; guards: a set of formula; tlabels: formula; actions: a set of set of proposition }
    Data: S, G, sig_guards: a set of proposition
    begin
    foreach node ∈ labeled_graph_node do
        fsm_node ← {name ← node.name; incoming ← node.incoming; guards ← node.guards; tlabels ← node.tlabels; actions ← ∅};
        for i in 0 to |node.guards| do
            fsm_node.actions ← fsm_node.actions ∪ {get_actions(node.guards(i))};
            fsm_node.guards(i) ← replace_actions_with_true(node.guards(i));
        fsm ← fsm ∪ {fsm_node};
        foreach fsm_node ∈ fsm do
            forall the A ∈ fsm_node.actions do
                S ← S ∪ get_emissions(A)
            (G, N) ← get_outgoing_guards_and_nodes(fsm_node,fsm);
            for i in 0 to |G| do
                sig_guards ← get_signals_in_guard(G(i));
                if sig_guards ⊈ S then
                    fsm ← fsm\{N(i)}
        fsm ← remove_accepting_node(fsm);
    return fsm

Merging equivalent nodes: In this part of the algorithm, lines 1 to 16, the nodes with the same label, i.e. nodes that satisfy the same proposition, are merged into a single node. The result of such a translation from Figure 4.9a is shown in Figure 4.9b.
4.1 Correct by construction code generation

Nodes \( N_{16} \) and \( N_{15} \) are updated with the edges of the nodes \( N_{11} \) and \( N_{8} \), respectively, since each of them satisfies the same propositional formula. Nodes \( N_{7} \) and \( N_{6} \) are also merged together since they both satisfy proposition \( st \) (c.f. Figure 4.9). Note that any outgoing edges to the accepting node \( N_{9} \) are removed after merging the nodes because their proposition is empty (\( \emptyset \)). For example, an outgoing edge from \( N_{11} \) to \( N_{9} \) is removed after merging with \( N_{16} \).

Removing connections to the Init node: This procedure is shown in Algorithm 4.1; lines 16 to 27. For each node in the graph, all the edges from the node Init are removed. In addition, all those initial nodes that are disconnected from Init are added to the set init_nodes for use in the next step. Figure 4.9c shows the resultant graph after removing Init in this step.

Updating sub-formulas: In the third step, which is shown at lines 28-32 in Algorithm 4.1, the algorithm searches for all the initial nodes (bar the single node with label \( st \)), that could not be merged in step-1, and tries to find nodes in the intermediate graph that satisfy these sub-formulas. If such nodes are found, the initial nodes are merged into these newly discovered nodes and their guards are updated, else the remaining nodes (and their paths to the accepting node) are discarded.

Reachability analysis: A simple reachability analysis is performed in this step (line 33 in Algorithm 4.1), which removes all the non-reachable nodes from \( st \). This process is further elaborated in Algorithm 4.2. The algorithm iterates through all the nodes in the graph and checks whether each node can be reached from \( st \) by recursively calling a function reachable_st (shown in lines 5 and 9-20). If the current node has no direct parent(s), i.e. node.incoming = \( \emptyset \), this function returns (1) \( \top \) if the current node is a starting node \( st \) (line 11-12), or (2) \( \bot \) if the current node is not \( st \) (line 13-14). Otherwise, the function recursively traverses the graph backwards, starting from the current node’s immediate parent(s) (line 16-19) until it returns \( \top \) or \( \bot \). Algorithm 4.2 then checks the final value returned from reachable_st at line 5. The result of this analysis is a set of nodes \( N \), which are reachable from \( st \) (line 7). For example, the path from \( N_{5} \) in Figure 4.9c is removed after this step.

Optimising the paths and extracting actions from guards: Next, all state transitions from any given node, guarded by the status(es) of internal signals, are checked whether they can ever be taken (line 34 in Algorithm 4.1). If guards are evaluated to false with respect to the emitted signals from the immediate predecessor state transitions, its paths to the accepting nodes are discarded. The detailed algorithm of this process is shown in Algorithm 4.3. Here, the input to the function is a set of nodes in the LGBA. The first part of this algorithm extracts the propositions
in `node.guards` that imply guarded actions on each incoming edge. A set of these propositions are then added to the set `actions`, which is a field of the newly created data-structure called `fsm_node` (lines 4-6). All the actions in `node.guards` are then replaced with boolean true (i.e. \( T \)). `fsm_node` is created for every node in the graph and added to a set `fsm` (line 8).

The next part of the algorithm collects all the signals emitted from the guarded actions `fsm_node.actions` into a set `S` (line 11). All the signals checked on the outgoing guards are also collected into a set `sig_guards` as shown at line 14. Then, if any one of the signals in `sig_guards` cannot be emitted from the previous transitions, i.e. \( \text{sig}_\text{guards} \not\subseteq S \), the subsequent node is removed from the set `fsm` (line 16). Lastly, the accepting node (e.g. `N9` in Figure 4.9c) is removed from the graph (line 14). The result of this translation is shown in Figure 4.10a, obtained from the compiler for the LGBA in Figure 4.8.

The generated Mealy automaton is an intermediate representation of the SC-SystemJ clock-domain, which can be compiled to: (1) a Promela process for verification by the SPIN model-checker or (2) C/Java code for execution. The verification step is not mandatory in the compilation process and can be bypassed. As an example, the generated C code is shown in Figure 4.10b for the Mealy automaton in Figure 4.10a. Edges in the automaton are annotated with `Guard Actions`. The transition is taken whenever the `Guard` is satisfied, and the corresponding `Actions` are performed. The generated C code is embedded inside another program; at a minimum, inside a machine interface that communicates with the environment.

Converting the generated Mealy automaton to C code is trivial. In Figure 4.10b, all the signal statuses are declared using the basic data types (lines 5-6) where `bool` is just a type synonym for `int`. The whole clock-domain logic is encapsulated in a function called `CD0` (lines 8-33). Calling this function results in a state transition of this clock-domain from one state to another. The current state of the clock-domain is represented as a label, whose address is stored as a value in the variable `CD0_L` (line 7). For example, suppose that the current state of the clock-domain is `N7` (line 23), then the program will jump to this label upon calling `CD0()` from the `main` function (line 36). Next, guards of the `outgoing` edges of the state `N7` are tested using `if` statements (lines 24 and 28). If the status of signal A is true, the corresponding action is performed which emits the output signal B (line 29). Otherwise, the program makes a transition with an empty action \( \emptyset \). Finally, the function returns after storing the label of the next state into `CD0_L` (lines 26 and 31).

Java code can also be generated from the final Mealy automaton in a very similar manner as C except that the current state of the clock-domain, e.g. `N7`, `N16` etc., is stored in a primitive variable rather than as a label. State transitions are then performed using
4.1 Correct by construction code generation

(a) Automaton generated from Algorithm 4.1 for SC-SystemJ program in Figure 4.3a

(b) C code generated from Figure 4.10a

Figure 4.10: Generated automaton and C code for SC-SystemJ example in Figure 4.3a

(a switch statement on this variable.

4.1.2 Generating a network of Mealy automata for GALS program

One important aspect of generating a network of Mealy automata for the SC-SystemJ GALS program is that all clock-domains are compiled individually. Therefore, generating a Mealy automaton of one clock-domain is independent from any others. These resultant sets of Mealy automaton are then put together as a network of Mealy automata that
communicate via channels. An example program with two clock-domains, communicating via a channel named $C$, is shown in Figure 4.11a and the corresponding compiled C code in Figure 4.11b.

The generated C code is a result of channel rewrite rules shown in Algorithm 3.1 for *send* and Algorithm 3.2 for *receive* statement, respectively. Since channel rendezvous is implemented using pure signals, $ACK$ and $REQ$ are also treated as signals and declared as $C_{ack}$ and $C_{req}$, respectively (lines 7 and 8). When the program first enters *send* $C$

```c
#include <stdio.h>
typedef int bool;
#define true 1
#define false 0

void *CD0_L;
void *CD1_L;
bool C_ack = false;
bool C_req = false;

void CD0(){
    if(CD0_L == NULL) goto N4;
    else goto *CD0_L;
    N21/*$3\_and\_not\_$1\_and\_not\_$2*/:
    CD0_L = &&N21; return;
}

void CD1(){
    if(CD1_L == NULL) goto N4;
    else goto *CD1_L;
    N12/*$5\_and\_not\_$4\_and\_not\_$6*/:
    if(!(C_req)) {
        C_ack = true;
        printf("Emitted :/uni2423C_{ack}\n");
        CD1_L = &&N12; return;
    }
    if(!((C_ack)) {
        C_req = true;
        printf("Emitted :/uni2423C_{req}\n");
        CD0_L = &&N14; return;
    }
    N4/*$st*/:
    if(true) {
        C_ack = false;
        CD1_L = &&N14; return;
    }
}

int main(){
    while(true) {
        CD0();
        CD1();
    }
}
```

Figure 4.11: Generating C code for SC-SystemJ example with a network of clock-domains
4.1 Correct by construction code generation

In the first clock-domain (CD0), it blocks until the presence of the signal ACK (lines 1-3 in Algorithm 3.1). This is shown in the compiled C code in lines 25-28 in Figure 4.11b. In this case, CD0 takes a self-loop transition, which results in re-entering the same state (as shown in line 27). Once ACK becomes true, CD0 makes a transition to N12 (lines 29-33). N12 denotes the second abort statement in Algorithm 3.1 (lines 4-7). While CD0 is waiting for the absence of ACK, it emits the signal REQ (line 16 in Figure 4.11b). Finally, CD0 terminates when REQ becomes false (line 22), and then settles in the state N21 (line 12-13). The behaviour of the second clock-domain (CD1) follows Algorithm 3.2 and its execution trace is almost identical to CD0.

4.1.3 Addressing specific issues

In this section we describe how our compilation strategy addresses classical issues that can occur in compilation of synchronous programs which are written in reactive languages such as Esterel [10] and SystemJ [21]. Herein, we use examples from standard literature on synchronous languages to describe the issues and their solutions.

**Instantaneous loops:** These can occur in the synchronous subset of SC-SystemJ programs. These are loops without any pause statement inside their body. As a result, once a clock-domain enters this loop, it cannot ever consume a tick or make a transition to another state. For example, while(true) emit S; is an instantaneous loop. If the induction rule for Inst(while(true) s) results in ⊤, it means that the body of the loop s can be terminated instantaneously without consuming any tick. Therefore, the compiler can statically detect instantaneous loops, by making sure that Inst transition of any loop is always ⊥.

**Causality:** As already explained in Section 2.5 and Section 3.3 causal problems do not occur with delayed signal communication semantics. This is ensured by the inductive rules in Figure 3.4 and Figure 3.5 where the emission of the signal, i.e. Inst(emit γ), only satisfies the proposition in the next instant Xγ. Furthermore, the inductive rules for all other signal checking statements, e.g. present and abort etc., are reduced to a logic which has only current time (i.e. non-X) propositions of the corresponding signal σ. Consequently, the generated SC-SystemJ transition system can only react on the signal emissions from the previous instant.

**Synchronous and asynchronous deadlocks:** In SC-SystemJ one can write static deadlocks – independent of input signals, and dynamic deadlocks – dependent on input signals. Example of dynamic asynchronous deadlock is given in Figure 3.11. An example of static synchronous deadlock is:
\{\texttt{await}(S1); \texttt{emit} S2;\} \mid \{\texttt{await}(S2); \texttt{emit} S1;\}

In the above example, the clock-domain can make no progress even though the program itself is “correct”. In this thesis, a singular approach has been taken to solve these problems – use of the SPIN model-checker with safety properties, i.e. verifying that something bad will never happen, specified as an LTL formula, to guarantee freedom from static and dynamic deadlocks. This approach is appropriate as opposed to performing such checks inside the compiler because complete state space analysis is still needed for detecting deadlocks, which is essentially model-checking anyway.

It is well known that the \textit{worst case complexity} of the generation of LGBA, and the resultant size, is exponential. This is due to the combinatorial explosion caused by synchronous parallel composition of states. Moreover, the method of semantic tableaux used to generate the LGBA tests for satisfiability of any arbitrary formula, by converting it into disjunctive normal form, can grow exponentially large. However, in this work, no resource constraint problems, especially running out of memory, are encountered during compilation because:

1. \textit{DNF transition rules}: The transition system \textbf{[Equation 3.3]} for each clock-domain and the corresponding inductive rules are already in disjunctive normal form, which directly supports the disjunctive partitioning of the transition relations.

2. \textit{Single non-nested next-time operator}: Since, the resultant LTL formula representing the transition system only ever contains non-nested next-time operator \(X\), the resultant LGBA can only ever have a depth of 2 \textbf{[73]}, which again significantly reduces the state space.

3. \textit{Storing only complete formulas in conjunction with tail-recursion}: A general tableaux method stores all rules solving satisfiability. The tableaux method described in \textbf{[73]} only stores the completely expanded (satisfied) formulas. The internal states generated when satisfying the formula can be discarded using tail-recursive procedures which reduce the exponential growth in stack space.

Large GALS programs can be compiled into Mealy automata without running out of memory by employing the aforementioned techniques. Nevertheless, the \textit{worst case} runtime complexity, and the resulting compilation time, grows exponentially with the length of the LTL formula \(\psi\) considered as a string. Fortunately, the transition system lends itself well to parallel processing. Each clause of the DNF representation of the transition system can be converted in parallel into the LGBA (as identified by \textbf{[74]}). Lastly, LGBA generation for separate clock-domains itself can also be carried out in parallel.
4.2 Verification of SC-SystemJ programs

SPIN \[44\] is a model-checker capable of checking propositional linear temporal logic (LTL) properties of distributed systems. These distributed systems are modelled in a language called Promela \[44\]. Promela processes are sequential processes that communicate via channels like those in CCS \[59\] or via shared variables. Channel communication halts the process when sending to a full channel or receiving from an empty channel, whereas shared variable communication never halts. For the compilation of SC-SystemJ channels, shared variables are used due to their logical tick-level blocking semantics as described in Section 3.5.2. In Promela, statements in different processes can be arbitrarily interleaved. However, this arbitrary interleaving can be controlled using atomic execution of statements. All statements encapsulated inside an atomic block are executed together. The SC-SystemJ compiler uses this atomic construct to emulate the SC-SystemJ big-step semantics and also to reduce the state space explosion problem when model-checking a network of clock-domains.

Usually, the model-checker needs to explore all possible program execution paths when verifying properties of a program. When verifying safety properties of a SC-SystemJ program, input signals can be left open and not closed by the plant model (Section 4.1). As a result, states in the generated automaton can have multiple outgoing edges with guards that check for different combinations of input signal statuses, e.g. whether an input signal \(A\) is present or not. Normally, the model-checker will explore all possible paths in the program using a backtracking algorithm. In Promela, backtracking is performed on if...fi statements in the case when more than one if condition is evaluated to true. Therefore, all input signals in SC-SystemJ programs need to be converted into simple truths, true, in the if conditions (guards) so that SPIN can traverse all paths in the generated state space using its backtracking algorithm. Any paths whose guards consist of non-emitted internal signals (and hence \(\bot\)) are automatically removed at compile time as explained in Section 4.1.1.

Referring back to the example in Figure 4.3a, the translation to Promela code from the corresponding Mealy automaton is very straightforward. Figure 4.12b shows the compiler generated Promela code for the Mealy automaton in Figure 4.12a (replicated from Figure 4.3a). Every transition in the Mealy automaton is translated into an atomic block in Promela (lines 6-10, 12-17, 19-24 and 26-33) with jumps (goto) to the next state as the final statement. The semantics of Promela dictates that the control can only switch from one clock-domain to another upon the completion of the atomic block. Similar to its C counterpart, every state transition from one state to another in the Mealy automaton is equivalent to one of the branches of the Promela if...fi statement. Hence, each if expression is the guard of the corresponding transition in the Mealy automaton. If
one of the if condition is evaluated to true, the corresponding action is performed such as emitting signals or executing data actions. When more than one condition is true, backtracking is performed as explained previously.

In the case of a network of clock-domains, channels are implemented as shared variables instead of using the built-in Promela message passing mechanism. This is mainly because Promela’s sending and receiving operations over the single queue channel completely halt the Promela process from running until synchronisation completes which is in contrast to the transition semantics in Equation 3.4. Furthermore, this hinders the implementation of a GALS system as demonstrated in the following example:

```
/* CD-S */

{ output channel P; send P;}
```
4.2 Verification of SC-SystemJ programs

```c
bool C_ack;
bool C_req;

active proctype CD0(){
goto N4;
N21/$3_and__not_$1_and__not_$2$:
   atomic {
      if :
         goto N21;
      fi;
   }
N12/$2_and__not_$1_and__not_$3$:
   atomic {
      if :
         if :: ((C_ack)) -> C_req = true;
            goto N12;
         :: (!((C_ack))) -> C_req = false;
            goto N21;
      fi;
   }
N14/$1_and__not_$3_and__not_$2$:
   atomic {
      if :
         if :: (!((C_ack))) -> C_req = true;
            goto N14;
         :: ((C_ack)) -> C_req = false;
            goto N12;
      fi;
   }
N4/$st$:
   atomic {
      if :
         :: (true) -> C_req = false;
            goto N14;
         :: (true) -> C_ack = false;
            goto N14;
      fi;
   }
}

active proctype CD1(){
goto N4;
N21/$6_and__not_$4_and__not_$5$:
   atomic {
      if :
         :: goto N21;
         :: fi;
   }
N12/$5_and__not_$4_and__not_$6$:
   atomic {
      if :
         :: (!((C_req))) -> C_ack = true;
            goto N12;
         :: (C_req) -> C_ack = false;
            goto N21;
      fi;
   }
N14/$4_and__not_$6_and__not_$5$:
   atomic {
      if :
         :: (!((C_req))) -> C_ack = true;
            goto N14;
         :: (!((C_req))) -> C_ack = false;
            goto N12;
      fi;
   }
N4/$sts$:
   atomic {
      if :
         :: (true) -> C_ack = false;
            goto N14;
      fi;
   }
}
```

Figure 4.13: Generated Promela code for SC-SystemJ GALS program in Figure 4.11a

/* CD-B */
{ input channel P;
  input signal A;
  output signal O;
  {receive P;}||{await(A); emit O;}
}

Here, CD-B tries to synchronize with CD-S through the channel named P. At the same time, CD-B also checks for the presence of the input signal A from the environment and emits O if A becomes available. Suppose this rendezvous is implemented using Promela’s built-in halting channel. When CD-B enters receive P, the entire clock-domain will halt until it completes the rendezvous with CD-S. As a result, CD-B may miss any incoming input signal A from the environment, which makes this program non-reactive thereby violating SC-SystemJ semantics.

Figure 4.13 shows the generated Promela code for the GALS program shown in Figure 4.11a. As one can see, the generated code is rather similar to the C version in Figure 4.11b. For example, ACK and REQ signals, used for implementing rendezvous, are declared as boolean variables. Furthermore, each state is represented as labels, and
the Promela process, i.e. clock-domain, can perform state transitions by jumping to one of these labels using the `goto` statement. In Promela however, processes run concurrently, and interleaving of clock-domains is automatically done by the SPIN scheduler at the boundaries of **atomic** blocks. Therefore, there is no need to store labels, unlike in C where they are needed for continuing the execution of previously executing clock-domains.

### 4.2.1 Equivalence of descriptions

In this section the proof of equivalence of the Promela model and the SC-SystemJ program is given using observational equivalence [59]. First, the following formal definitions of the SC-SystemJ transition system are given:

**Definition 10. Run:** Let $M$ be a transition system. Then a run of $M$ over the signal $2^S$, where $S = I \cup O \cup Y$, is defined to be an infinite sequence of states $\pi = q_0 \xrightarrow{s^0} q_1 \xrightarrow{s^1} q_2 \ldots$, where $S^i \subseteq S$, $i \geq 0$.

**Definition 11. Language:** Let $M$ be a transition system. Then the language of $M$, denoted $L(M)$, is meant to be the set of all possible runs of $M$.

Let observers $O_p$ and $O_s$ observe the transition of the Promela process and the SC-SystemJ clock-domain, respectively. Let $S_s$ denote the set of both input and output signals for the interface of the SC-SystemJ clock-domain as well as the internally declared signals. A clock-domain observer is defined as a transition system $O_s$ over $S_s$ that can observe any possible run over $S_s$. The observer $O_s$ is synchronously composed with the transition system of the SC-SystemJ clock-domain and observes after each synchronous reaction, sampling all the signals in $S_s$. Similarly, $S_p$ is defined as the set of variables in the Promela process that corresponds to $S_s$ in the SC-SystemJ clock-domain. An observer $O_p$ of the Promela process is defined as a transition system over $S_p$ that can observe any possible run over $S_p$. The observer $O_p$ is composed synchronously with the Promela process, observing only after the completion of the Promela process transition, defined at the boundaries of the **atomic** blocks. Indeed, the observer cannot observe inside the atomic section. Now the following can be stated *under the assumption that the SC-SystemJ program is correct*, i.e., each clock-domain is reactive and deterministic.

**Definition 12. Trace Equivalence:** Let $M$ and $N$ be two transition systems with the same alphabet. $M$ and $N$ are trace equivalent, denoted $M \approx N$, if they have the same language $L(M) = L(N)$.

**Lemma 4.2.1.** Let $M = (Q_1, s_{t1}, I_1, O_1, Y_1, A_1, T_1)$ and $N = (Q_2, s_{t2}, I_2, O_2, Y_2, A_2, T_2)$, be a clock-domain and its Promela translation, respectively. Let $O_1$ and $O_2$, be the observers for $S_s = I_1 \cup O_1 \cup Y_1$ and $S_p = I_2 \cup O_2 \cup Y_2$ on $M$ and $N$, respectively, and $L_1$
and $L_2$ be the languages they observe via $S_s$ and $S_p$. Then the following holds: $L_1 = L_2$ and by Definition 12, $M \approx N$ with respect to $O_s$ and $O_p$.

Proof. Since the model of the SC-SystemJ clock-domain is contained within the Promela process and invoked from inside the Promela process, the proof is by construction. The two observers $O_s$ and $O_p$ observe observables (words) with the same alphabet, i.e., $s \in L(M) \iff s \in L(N)$. First, we need to prove that $s \in L(M) \iff s \in L(N)$. $O_s$ can only observe signals in $S_s$ generated by $M$ at the end of the tick, which is an atomic transition (due to big-step semantics). These global variables are updated inside the atomic block and are observable only at goto where they remain unchanged in Promela hence, $S_p = S_s$ at the end of tick. Next, we prove $s \in L(N) \implies s \in L(M)$. $O_p$ observes only global variables (in $S_p$) produced by $N$, moreover, $O_p$ cannot observe any internal transition of the atomic block. Furthermore, non-execution of $N$ does not change the set $S_p$ and hence, infinite runs of a Promela program without execution of $N$ in between two runs of $N$ is equivalent to two consecutive runs of the $N$ and hence, $s \in L(M) \iff s \in L(N)$.

\[\Box\]

Theorem 4.2.1. Let $M = \{M_1, M_2, \ldots\}$ and $N = \{N_1, N_2, \ldots\}$, be a transition system for a network of clock-domains and its Promela translation, respectively. Let sets $O_S = \{O_{s1}, O_{s12}, \ldots\}$ and $O_P = \{O_{p2}, O_{p21}, \ldots\}$, be the observers for $M$ and $N$, respectively and sets $L_1 = \{L_1, L_12, \ldots\}$ and $L_2 = \{L_2, L_21, \ldots\}$, be the languages they observe. Then the following holds: $\bigwedge_{k \in |L_1|} L_1_k = L_2_k$ and by Definition 12, $M \approx N$ with respect to $O_S$ and $O_P$.

Proof. Proof follows from Lemma 4.2.1. \[\Box\]

It is worth mentioning that the above proofs hold in the absence of fairness guarantees. But, since the interface signals in a set $S_p$ do not change for a non-executing run of $N$, one cannot guarantee rendezvous without fairness. Hence, weak fairness \[\Box\], i.e., every process that is almost always enabled should be executed infinitely often, is an essential (but not complete) condition for a successful rendezvous completion.

### 4.2.2 Functional properties to be ensured for the ice cream manufacturing facility

The generated Promela code can be verified against a set of properties specified as an LTL formula. The formula can have the usual LTL boolean and temporal operators as well as SC-SystemJ specific propositions such as ‘signal $A$ is emitted’, ‘the value of the signal $A$ is 5’, etc. In this section, the automated ice cream manufacturing facility, described in
Section 2.3 is used as an example to show how functional properties can be specified as LTL formulas and proven for the SC-SystemJ control-logic.

Validating that the system will behave according to the design specification after its deployment, and verifying that the system is semantically sound are crucial elements in the design of any safety-critical system. In the ice cream manufacturing example there are several key functional properties that must be checked in order to avoid potentially incorrect behaviour. Here, five such properties are presented in order to demonstrate the verification approach used for verifying SC-SystemJ programs. In this case, it is assumed that the system will not miss any input events during its operation (this can be guaranteed using static analysis [17]:

Property 1: When the ice cream arrives, it should always eventually be carried to the storing station when there are no faults on the conveyor belt.

LTL Formula:

$$G(\neg Path1Fault \land \neg Path2Fault) \land G(\neg Path1Fault \land \neg Path2Fault) \land G(ICDetect \rightarrow F(enSt1 \lor enSt2))$$

During normal operation, in the absence of any fault, the system should process the ice cream following Path 1 or 2 depending on the customer’s order. That is, the system should always eventually emit either enSt1 or enSt2 upon reception of the signal ICDetect.

Property 2: There should be no fault on the conveyor belts until the ice cream enters the ICMF. If the conveyor belt for Path 1 is faulty at the time when the ice cream is detected by the ICMF, the system should and eventually route the ice cream to Path 2.

LTL formula:

$$\left(\neg Path1Fault \land \neg Path2Fault\right) \lor \left(G(\neg Path1Fault \land \neg Path2Fault) \land G(ICDetect) \rightarrow F(enSt2)\right)$$

When Path 1 is faulty, the ice cream should not be sent to Path 1. In this case, the system should always re-route the ice cream to Path 2 regardless of the options that the customer had chosen.

Property 3: There should be no fault on the conveyor belts until the ice cream enters the ICMF. If the conveyor belt for Path 2 is faulty at the time when the ice cream is detected by the ICMF, the system should and eventually route the ice cream to Path 1.
Property 4: When both Paths 1 and 2 are faulty the system generates ERROR and DPLACE signals in order to halt the entire system

LTL formula:

\[
\left( \neg \text{Path1Fault} \land \neg \text{Path2Fault} \right) \cup \left( \neg \text{Path1Fault} \land \neg \text{Path2Fault} \right)
\]

Similar to Property 2, but in this case Path 1 should always be chosen.

The system should not let any items be carried to Path 1 or Path 2 when both paths are faulty. Therefore, in such circumstances, the system should always eventually generate the ERROR signal in order to stop the conveyor and DPLACE to stop the ice cream placing station.

Property 5: The system should always be deterministic for the generation of the signals for PLACE, DPLACE, and enSt1, enSt2 such that they should not be emitted in the same tick

LTL formula:

\[
G\left( \neg \text{enSt1} \land \neg \text{enSt2} \land \neg \text{PLACE} \land \neg \text{DPLACE} \right)
\]

This property checks whether the system produces deterministic output events. Informally, the fault detection and recovery controller (FDRC) should not generate both request (PLACE) and stop (DPLACE) signals at the same time (see Figure 2.6), and the storing station controller (SSC) should not try to pick up the ice cream from both paths 1 and 2 at the same time.

Properties 1-4 are so called liveness properties [55]; guaranteeing that something good will eventually happen. Property 5, on the other hand, is a safety property [55], guaranteeing that something bad will never happen. In this particular example, these liveness properties require a plant model for verification because they are dependent on the control logic being triggered by input signals (e.g., ICDetect) at certain points of the program execution. On the other hand, Property 5 is invariant; it should hold irrespective of the plant presented to the control logic. Hence, the environment of the control logic is closed
(as shown in Figure 4.14) for liveness property verification, while the environment is left completely open so that the model-checker explores all possible paths of the control logic when verifying the safety property (invariant).

Figure 4.15 shows a snapshot of iSPIN (GUI version for SPIN) performing verification for Property 5. Here, the fairness constraint under the liveness panel is set (1), which tells SPIN that the Promela processes (i.e. clock-domains) should run under fairness assumptions. This prevents starvation of processes during the verification process. SPIN is set to use the (LTL) claim for verification (2), which checks whether the system conforms to the specified LTL property. (3) shows the LTL formula proven on the Promela process. The LTL formula is the same as Property 5; it is just expanded to the basic components by the SPIN model-checker. Finally, the verifier shows the satisfaction of the LTL property in the output console (4).
4.2 Verification of SC-SystemJ programs

The number of visited states indicates the amount of work done by SPIN in order to complete the verification process. It is shown that SPIN traversed the largest number of states (24,066) while verifying Property 2, whereas it traversed the smallest number of states (8,357) while verifying Property 1. The verifier took less than a minute to check each property.

Open vs closed loop system verification

There are two types of verification techniques for verifying a reactive system developed in the SC-SystemJ language: open and closed loop techniques. As already explained in Section 4.2.2, the closed-loop verification approach is used for verifying Properties 1-4 of the ice cream manufacturing facility example, whereas the open-loop approach is used for verifying Property 5. Properties 1-4 can also be verified using the open-loop verification technique in which there is no need to combine the controller logic with the plant model. One advantage of the open-loop verification is that a plant does not need to be updated for verifying different LTL properties. However, this approach has two drawbacks:

1. Increase in the complexity of the verification: A plant model defines the emission of the input signals at specific points of the program. For instance, Figure 4.14a shows a plant for verifying Property 1 where the signal A is emitted in the second clock-domain tick. In the case of open-loop verification, signal A can be present or absent at any tick, and the model-checker verifies all possible choices. However, the same
property used for verification in the closed-loop system cannot be used in open-loop verification because neither the program nor the LTL property specifies when signal A is being emitted; this is specified in the plant model. Consequently, this information needs to be incorporated in the LTL property, which inevitably results in increasing the length of the LTL formula. It is well known that the complexity of LTL model-checking is exponential to the length of the LTL formula \[53\]; open-loop verification can take longer depending on the properties to be verified.

2. Expressiveness of the plant model: As explained above, designers need to write more complex LTL properties which can be difficult when the system being verified operates in a very complex environment. In closed-loop system verification, SC-SystemJ programmers can easily model a plant using the same language.

On the other hand, open-loop verification is often useful when verifying safety properties. In this case, the properties are specified such that the system should always not be in specific states, or certain signals should never be emitted, etc., regardless of the input event patterns generated by the environment. The next section introduces additional benchmark programs as well as the results obtained via open and closed-loop verification approaches.

4.3 Experimental results for other benchmark programs

In this section, a set of SC-SystemJ benchmark programs is used in order to characterise the experimental performance while verifying LTL property specifications on various types of GALS programs. The first experiment measures the compilation time, size of SC-SystemJ source code and results of functional verification via SPIN. Next, the executable code generated from the automata and AGRC-based SystemJ compilers are compared in terms of size and average execution time between the start and the end of the clock-domain ticks, i.e. average reaction time of clock-domains. The examples include a washing machine controller \[76\], conveyor controller \[76\], robot motion controller \[76\], and a dual-chamber pacemaker \[77\]. All these examples are converted into SC-SystemJ from their original description in CRSM \[76\] or UPPAAL \[77\]. Promela code generated from the SC-SystemJ compiler is then verified against liveness properties or safety properties. The LTL properties specified are shown in Table 4.2. The first three properties in the table are liveness properties, while the others are safety properties. A procedure for verifying liveness properties is identical to the ones presented in Section 4.2.2: a plant model is synchronously composed (||) with the control logic to generate input stimuli for the system,
and the translated automaton is verified in SPIN. On the other hand, input signals are left completely open for verifying the safety properties.

### 4.3.1 Benchmark descriptions

The washing machine controller example is a pure synchronous program that consists of two main reactions; the washer and the drier. Each reaction awaits for an input signal from a user which activates its washing or drying operation. The property verified on the system is that both reactions never activate at the same time, i.e. signals \textit{WASHING} and \textit{DRYING}, which indicate the start of washing and drying operation respectively, are not emitted in the same instant. The robot motion controller consists of a camera and motion controllers that continuously locate an object, move to its location, and, then picks up that object. The system consists of two clock-domains, one for each controller. The property verified on the system is that the object should be picked up before receiving the next object’s location. The conveyor control system consists of two mechatronic components, a conveyor, and a mechanical arm, each modelled as a SC-SystemJ clock-domain. These are connected together into a mechanism for sorting items. Once an object is placed, the first clock-domain sends this information to the second clock-domain which moves the conveyor belt until the object is displaced off the conveyor by the mechanical arm. In this case, it is checked whether the system correctly counts the number of items on the conveyor when the sender and receiver exchange messages.

A dual-chamber implantable pacemaker is a case-study in safety-critical system design using SC-SystemJ and is adopted directly from the one modelled by in UPPAAL. A pacemaker is a small device that monitors a patient’s heart rate and artificially generates electrical pulses when it does not detect a heartbeat within a set time period. As shown in Figure 4.16a, the heart has two major types of tissues which governs its electrical conduction system and controls the cardiac cycle. The Sinoatrial (SA) node, which is located in the upper wall of the right atrium, periodically generates electrical pulses which cause
both the atria to contract, allowing blood to enter the ventricles. The Atrioventricular (AV) node delays the electrical pulses from the SA node to allow the ventricles to be fully filled before both chambers contract to pump blood out of the heart. The two main components of the pacemaker are shown in Figure 4.16b.

The pacemaker developed in SC-SystemJ is able to sense two types of activities called Atrial Sense (AS) and Ventricular Sense (VS) from the SA and AV nodes, respectively. They indicate activation of these two heart tissues that control the rate of blood pumping out of the heart. A heart disease or degraded functionality of the heart due to aging can cause abnormal heart rhythms which may result in either insufficient or too much blood supply to the body. When the heart operates normally, a pacemaker receives AS and VS one after another within a predetermined period. Whenever the pacemaker detects absence of VS or AS within a certain time, it generates artificial pulses called Ventricular Pacing (VP) or Atrial Pacing (AP). Lower Rate Interval (LRI) component is responsible for generating AP after sensing a ventricular event (VS or VP) if AS is not generated by the heart after certain time. Conversely, Atrio Ventricular Interval (AVI) component generates VP after an atrial event (AS or AP) if VS is not generated by the heart after a certain time period. In order to check the correctness of the design, the system is verified against LTL properties in order to guarantee generation of the event VP and AP when AS or AP and VS or VP are not detected within a deadline, respectively.

4.3.2 Verification results for the benchmark programs

Collected data from the verification of the SC-SystemJ models for the pacemaker and other examples are shown in Table 4.2 and Table 4.3. All benchmark programs were run on a desktop machine with the following specifications: Intel Core i5 CPU 660 @

\footnote{This image was obtained from: http://www.texasheart.org/HIC/Topics/Proced/icdtopic.cfm}
4.3 Experimental results for other benchmark programs

Table 4.3: Performance of the compiler

<table>
<thead>
<tr>
<th>Example</th>
<th>Time to compile (s)</th>
<th>Size (lines)</th>
<th># of CDs</th>
<th># of reactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacemaker</td>
<td>118.76</td>
<td>108</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Robot motion controller</td>
<td>0.182</td>
<td>32</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Washing machine controller</td>
<td>2.29</td>
<td>35</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Conveyor</td>
<td>0.172</td>
<td>39</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

3.33Ghz, 4GB memory and Windows 7 64-bit operating system. The pacemaker consists of two clock-domains (LRI and AVI) each containing three main reactions. This example contains a large number of nested conditional and pause statements that resulted in an increased number of states as well as possible state transitions in the system. It took 118.76 seconds to compile this example. The verification report generated by SPIN showed that the example consists of a total number of 62 states, and it needed to perform 248 transitions to verify the first LTL property of the pacemaker as shown in Table 4.2, whereas only 160 transitions were needed to be traversed in order to verify the second property. These significant reductions in the number of states and transitions can be attributed to the atomic construct, without which the same behaviour requires 16369 states, 106115 transitions, and 14253 states, 56025 transitions for verifying each LTL property, respectively. The robot motion controller and the mechanical conveyor/arm examples are also two clock-domains systems, but they are much simpler in terms of complexity of control-flow, and hence result in both fewer number of states and transitions to verify the LTL properties by SPIN. Each of these SC-SystemJ program is 32 and 39 lines long and takes 0.182 and 0.172 seconds to compile, respectively. This reduction in compilation time is due to the fact that there are fewer number of parallel reactions in each of these examples resulting in lesser compilation effort in creating the parallel composition of states (Section 4.1.3). It is interesting to see that there are more states and transitions needed in verifying Conveyor (72 and 210) than Robot (27 and 55). This is because the environment is completely open for verifying the safety property in Conveyor, while closed for verifying the liveness property in Robot. On the other hand, the washing machine controller, which is a pure synchronous program, took more time to compile (2.29 seconds) compared to the Robot and the Conveyor examples since it uses more parallelism in the program. However, in this example, there are fewer number of states as well as transitions required for verifying the property (Table 4.2). This was expected as there is no asynchrony in this example; asynchrony would have increased the complexity for verifying the properties in SPIN. Still, these numbers are comparable to that of Robot since the open-loop verification approach was also used in this example (i.e. safety property).
4.3.3 Generated back-end code size, runtime, and compilation time comparisons

Figure 4.17a shows the comparison of executable code sizes generated from the automata (SC-SystemJ) and AGRC (SystemJ) compilers. All the benchmark programs are first

![Comparison of Code Sizes](image)

- **Washing**: 4.67 KB (AGRC-based), 1.4 KB (automata-based)
- **Conveyor**: 8.6 KB (AGRC-based), 2.7 KB (automata-based)
- **Robot**: 12.7 KB (AGRC-based), 2.9 KB (automata-based)
- **Pacemaker**: 16 KB (AGRC-based), 6.3 KB (automata-based)

![Comparison of Average Reaction Times](image)

- **Washing**: 573 ns (AGRC-based), 230 ns (automata-based)
- **Conveyor**: 311 ns (AGRC-based), 189 ns (automata-based)
- **Robot**: 310 ns (AGRC-based), 180 ns (automata-based)
- **Pacemaker**: 688 ns (AGRC-based), 221 ns (automata-based)

Figure 4.17: Comparison between the AGRC-based and the automata-based compiler in terms of generated code size and average reaction times
4.3 Experimental results for other benchmark programs

Compiled into Java source code, which are then converted into the class files using the version 1.8 of the Java compiler. Note that the size of the SystemJ Run-Time Support (RTS) library \cite{30, 49} is excluded from these figures for fairness of the comparison. The automata compiler generates on average 3.3x smaller codes than the AGRC compiler. There are two main reasons for this: (1) In the new automata semantics, control-flow of a program is just state transitions from one state to another, and are explicitly specified by programmers using the \texttt{pause} statements. On the other hand, the AGRC semantics defines a sequence of fine grained micro-steps for individual SC-SystemJ statements between \texttt{pause} statements. As a result, the AGRC compiler generates additional code for these micro-step executions, which leads to overall increased code size. (2) In addition, the automata compiler optimises the generated code by removing unreachable paths in the FSM. As explained in \ref{section:4.1.1} statically determining the status of internal signals in any FSM states is trivial in the automata semantics. For instance, for each state in the FSM, the compiler checks signal emissions (i.e. propositions) in all actions on the incoming edges. Then, if there is an outgoing guard for signals that cannot be emitted based on the previous actions, the corresponding path is removed from the FSM, resulting in reduced generated code size.

Figure 4.17b shows the average reaction time of clock-domains in the benchmark programs. Reaction time indicates the time taken for a clock-domain to execute between two consecutive ticks. This was measured on a Core i5 CPU 660 3.33Ghz desktop with 4GB RAM using the Java program profiler called VisualVM\cite{78} in such a way that the total execution time of a Java method for a clock-domain tick is divided by the number of invocations. Furthermore, the presented results are collected based on one million clock-domain ticks. As one can see from the figure, the average reaction times of the programs compiled using the automata compiler are faster, on average by 2.1x, compared to the program from the AGRC compiler. Again, this is because the AGRC compiler generates fine grained control-flow for the clock-domain execution (micro-step). Moreover, in the AGRC-based approach, status and value of the signals are accessed via getter and setter method calls, whereas in the automata-based approach, they are more efficiently accessed via direct reference to the Java class fields.

The compilation time comparison is shown in Figure 4.18. Both the AGRC and the automata compilers generate a target source code (C or Java) for all the benchmark programs within a reasonable time. In particular, the automata compiler outperforms the AGRC compiler in Washing, Conveyor, and Robot benchmarks whereas the AGRC compiler is \textasciitilde 2.5x faster for Pacemaker benchmark. This is mainly due to that the automata compiler has to perform a large number of equivalence checks on LTL formulas for generating an LGBA as the program size increases. The AGRC compiler, on the other hand, does not need to do such checks and this lead to faster compilation time for the bigger
In this chapter, a novel approach for generation of What You Prove Is What You Execute (WYPIWYE) Globally Asynchronous Locally Synchronous (GALS) programs designed in SC-SystemJ was presented. The approach produces a network of Mealy automata from SC-SystemJ programs automatically, which are then translated into Promela for verification using Linear Temporal Logic (LTL) properties, or into executable C/Java code. The presented approach is scalable which has been achieved by first translating SC-SystemJ programs into LTL formulas consisting of non-nested next time operator ($X$). This reduces the amount of memory required to create Labelled Generalized Büchi Automata (LGBA) from a transition system described in an LTL formula. A novel algorithm is then applied to build a network of deterministic Mealy automata, from which back-end code can be generated. Lastly, the SC-SystemJ compiler generates both smaller and faster executable than the original SystemJ compiler.
The compilation and verification techniques presented in Chapter 4 enable verification of functional correctness properties of SC-SystemJ programs. Yet, this approach does not include analysis of the system’s ability to meet timing requirements in a real-time setting. This chapter presents analysis and scheduling of GALS programs to bound response times to input events. The proposed approach is applicable to scheduling of GALS programs for different target architectures with single or multiple cores, with shared or distributed memory.

Real-time systems are computer systems that must complete their computation and deliver the result within strict time constraints which are imposed by their environments. Therefore, the correct behaviour of such systems depends not only on the result of the computations, i.e. functional correctness, but also on the physical time at which these results are made available to the environment. There are two major approaches for designing real-time systems: the real-time schedulability analysis approach [79, 80, 81] and the reactive language approach [10, 11]. The real-time schedulability analysis approach to real-time system design concentrates primarily on meeting deadlines of all tasks in the application; a technique for verifying functional correctness of programs is not tightly integrated. Reactive languages, on the other hand, are based on formal mathematical semantics [10, 11, 40]. Formal semantics allow tight integration of techniques to guaran-
functional correctness and real-time bounds. There has been significant effort within the reactive language community [82, 83] to build tools that automatically provide timing guarantees. All current work on real-time analysis of reactive applications concentrates on obtaining tight (i.e. close to the actual) values of worst-case execution time of applications (termed worst-case reaction time (WCRT) [17] in the reactive language community). In the case of purely synchronous reactive languages such as Esterel [10] and Signal [11], the WCRT value alone suffices to guarantee that: (1) no incoming input events from the environment are missed, provided the WCRT value is shorter than the minimal inter-arrival time of input events and (2) the output events generated in response to the input events are bounded by the WCRT time, due to the synchrony hypothesis, which states that the reaction to incoming events is carried out in zero time and outputs are produced instantaneously. Although programs designed in synchronous reactive languages can be formally analysed for functional correctness guarantees and real-time bounds, these languages are unsuitable for designing distributed systems since all parallelism in these languages is compiled away to produce a single sequential program [56]. To address this, there have been efforts targeted at desynchronising [84] synchronous programs. These works guarantee that a desynchronised program has the same functional correctness guarantees as the original synchronous program, but make no claims about the real-time equivalence of the synchronous program and its desynchronised version. In fact, to guarantee real-time properties, a synchronous program needs to be reanalysed after desynchronisation. The work presented in this chapter targets building real-time distributed and shared memory systems using SC-SystemJ. Moreover, it can complement the work on desynchronising synchronous programs [84] by guaranteeing their response time bounds.

Traditional real-time scheduling theory [80] is not directly applicable to the GALS approach. This is because some of the fundamental characteristics, taken for granted, by the real-time task scheduling community [80, 81] are missing in the GALS model. First of all, there is no notion of a deadline for clock-domains; their ability to capture incoming events is simply determined by the WCRT. Therefore, in order to apply real-time scheduling theory to the GALS model, one needs to somehow relate the WCRT to a period or deadline of a real-time task. For example, if one considers the WCRT of a clock-domain to be its period, and for the sake of simplicity, also its deadline, then assuming atomicity of clock-domain execution (i.e. they are non-preemptible within their logical tick by definition), cyclic scheduling of clock-domains is the most straightforward way to minimise WCRT if multiple clock-domains are running on shared resources. This is because the order of clock-domain execution is inconsequential since WCRT is formally defined as the worst possible time between the start of the current and the next clock-domain ticks [85]. However, a cyclic schedule that shortens the WCRT does not necessarily guarantee the response time bounds of GALS programs. Here, response time in a GALS system is
defined as the time taken for the system to run one or more clock-domains’ ticks in order to generate one or more desired final output event(s) in response to the corresponding input event(s). Finally, most if not all classical real-time scheduling algorithms \([80, 81]\) are based on guaranteeing that the task response time – time from release to completion is within the specified deadline. The applicability of classical real-time scheduling approaches to our definition of response time, while adhering to the GALS model of computation, is at best doubtful. There does exist work \([25]\) that provides a library of function calls that can be used to build programs following the GALS model of computation on top of a real-time operating system. However, enforcing the GALS model and its semantics is left up to the programmer rather than the compiler like in SC-SystemJ. Since one cannot enforce GALS semantics at compile time, statically proving functional correctness and real-time guarantees of the developed programs is impossible; providing these guarantees is the target of the work presented in this chapter.

More recently authors in \([86]\) presented a technique to estimate, at system level, worst-case communication latency between processing elements in network-on-chip (NoC) architecture. While their approach enables measuring of accurate latency upper bound by employing a formal verification method, they limit the model to a synchronously clocked NoC, i.e. a single global clock drives all the NoC routers. Furthermore, their approach focuses more on verifying latency of packets with a given topology and routing information in a NoC setting while the work presented in this chapter targets scheduling GALS programs to bound response times to input events.

The major challenge in analysing response times of GALS system arises from their expressiveness. Each clock-domain itself forks reactions that run in synchronous parallel and in lock-step with the clock-domain tick. In addition, reactions can themselves be composed of other parallel reactions to an arbitrary depth. Moreover, a rendezvous (message-passing) communication protocol used to exchange data between reactions in different clock-domains can be carried out in parallel and preempted at tick boundaries by the environment, which further complicates the static response time analysis problem. Finally, the standard blocking rendezvous communication protocol \([7]\) is unsuitable for guaranteeing bounded response times, because the blocking rendezvous communication model is unbounded in time. Hence, a new rendezvous communication model needs to be incorporated in the response time analysis.

The major contributions of the work presented in this chapter are: (1) definition of response time in a GALS setting where multiple synchronous and asynchronous tasks are interacting to generate output events as a response to input events, (2) a Satisfiability Modulo Theory (SMT) \([46]\) based scheduling approach for GALS systems to guarantee response time constraints, and (3) a rendezvous communication model for GALS systems amenable for real-time scheduling.
Section 5.3 presents a way to specify the response time constraints in GALS setting using the motivating example. Section 5.4 describes the design flow and the SMT formulation for guaranteeing response time constraints. Section 5.5 gives the complexity analysis of the proposed SMT approach to scheduling. Finally, Section 5.6 provides benchmark results that show that the generated schedule for the clock-domains meets the response time constraints, as well as scalability of the approach in a multi-processor distributed memory system.

5.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding satisfiability of logical formulas. SMT is an extension of the boolean satisfiability problem, also known as SAT. An objective of SAT is to find whether there are possible solutions (models) for the boolean variables (true or false) which results in the overall formula being evaluated to true. SMT extends SAT with a richer modelling language by providing theories and basic syntax and grammar rules for describing problems. In SMT, interpretation of variables is constrained by some background theories. For example, the integer arithmetic theory restricts interpretation of variables to comparison functions ($<$, $>$, $<=$, $>=$), integer numerals, and the usual arithmetic operators ($+$, $-$, $*$, mod, div) \[87\]. Furthermore, one or more theories can be combined to define logic such as QF_LIA, which includes both boolean and linear arithmetic theories.

Logical formulas in SMT can be described in first-order logic. However, the SAT problem, which is a subset of SMT, is NP-complete, and first-order logic is undecidable \[88\]. Therefore, SMT solvers primarily focus on proving a small subset of the formula, which effects satisfiability rather than validity of a formula, i.e. finding a solution to a set of constraints rather than proof of a statement \[87\]. Response time analysis, presented in this chapter, is also a satisfiability problem, i.e., it determines if there is at least one possible schedule which guarantees the system's response to specific input events within some specified time bound.

To illustrate the use of SMT, consider a mixed shop scheduling decision problem \[89\]. Assume there are two types of jobs: $f_i$ and $o_i$, where $f_i$ implies flow and $o_i$ implies open shop type jobs. Each of these jobs should be processed on two machines $M_1$ and $M_2$. In addition, once a job is released, it has to complete processing before starting to process on
the other machine. Flow shop type jobs have to be processed consecutively, first on $M_1$, then on $M_2$. On the other hand, open shop type jobs can be released on these machines in any order. Figure 5.1a shows an overview of processing times for two flow shop type jobs ($f_1$ and $f_2$) and one open shop type job ($o_1$). The objective is to find a schedule that will complete the processing of all jobs within 9 time units ($T_{max} \leq 9$). In order to solve this problem in SMT, all constraints are encoded as an inequality, for example, $x_{i,k} \geq x_{i,j} + d_{i,j}$. In this constraint, $x_{i,j}$ denotes start time of a job $x_i$ on a machine $j$ where $j \in \{1, 2\}$, and $d_{i,j}$ is the corresponding processing time. Therefore, this inequality states that the start time of $x_i$ on $M_k$ is at any time greater than the start time of $x_i$ on $M_j$ plus its duration $d_{i,j}$.

Figure 5.1b shows four types of constraints encoded in SMT formula. First, all jobs should be released at any time greater than or equal to 0. These are encoded as $x_{i,j} \geq 0$ in Figure 5.1b(i). Next, Figure 5.1b(ii) specifies precedence constraints. For example, the first clause indicates that $f_{1,2}$ can be released after processing $f_{1,1}$. On the other hand, logical disjunction ($\lor$) in the third clause states that open shop type jobs can be released in any arbitrary order. To make sure only one job is processed at a time on each machine, resource constraints are specified in Figure 5.1b(iii) and (iv) for flow and open shop type jobs, respectively. For example, the formula in the form of $(x_{k,j} \geq x_{i,j} + d_{i,j}) \lor (x_{i,j} \geq x_{k,j} + d_{k,j})$ in (iii) states that jobs $x_i$ and $x_k$ do not overlap on $M_j$. For open shop type jobs, this constraint is extended such that they do not overlap with other jobs on both
machines. Finally, end time constraints are specified in Figure 5.1b.

An execution trace of jobs satisfying $T_{\text{max}}$, according to the schedule generated from the Z3 SMT solver, is shown in Figure 5.2. As can be seen, $f_{1,1}$ is started first at time 0 on $M_1$ followed by $f_{2,1}$. Since no other flow shop type jobs can be started, $o_{1,1}$ is also started simultaneously on $M_2$ at time 0. $M_2$ then becomes idle until $M_1$ completes processing $f_{1,1}$. Jobs $f_{2,1}$ and $f_{1,2}$ start processing on $M_1$ and $M_2$, respectively, at time 2. Finally, both $o_{1,2}$ and $f_{2,2}$ start at time 5. The end time constraint is satisfied by finishing the job $f_{2,2}$ at time 9.

As illustrated so far, SMT is well-suited for finding a schedule of tasks that satisfies a given set of constraints expressed using inequalities. In the next section, the ICMF case-study is revisited in order to define the response time problem, which will later be formulated as an SMT problem.

5.2 Motivating example

In order to illustrate the response time of a system, consider again the automated ice cream manufacturing facility (ICMF), which was presented in Section 2.3. In this chapter, a simplified version of the original ICMF is presented to describe the real-time problem and present its solution. Although the methodology introduced here is still applicable to the original scenario, this new example omits some of the functionality in the original version in order to focus more on the fundamental concepts of this chapter and also for brevity of discussion.

Figure 5.3 shows that the ICMF unit consists of conveyor belts, with photo-eyes and a tag reader for ice cream passage detection, a turntable with sauce and sprinkle dispenser, and a diverter to divert the ice cream to the correct path. The basic operation of this ICMF unit is identical to the original design; when the customer’s order includes a selection of toppings the diverter is triggered to route the ice cream to the turntable (Path-2 in Figure 5.3), or to the alternate path (Path-1 in Figure 5.3) when the order does not include the toppings. The fault-recovery facility is omitted in this simplified version, hence the fault-detection and recovery controller (FDRC) is renamed to photo-eye and tag reader controller (PEC). Additionally, the storage station controller (SSC) in the original design
is not included in this example, and the ice cream is expected to enter this manufacturing unit without needing to request the placing station to place an ice cream on the conveyor.

Figure 5.4 shows the simplified control logic for the ICMF unit, with three clock-domains allocated to the three separate processors, PEC, DC, and TTC, connected to the photo-eye and tag reader, the diverter, and the turntable, respectively, as in Figure 5.3. The clock-domain implementing PEC waits for an input signal A (line 11) from the photo-eye and tag reader, which indicates the presence of an ice cream on the conveyor. During the same tick, PEC informs a monitoring computer about the incoming ice cream by emitting O1, and also performs data computation by calling `extern process(#A)` in line 14. If the value on the tag is 0, the customer has asked for toppings, and this information is conveyed to both DC and TTC, via rendezvous on channels CH1 and CH3 (in lines 16 and 20). Between these channel communications there is another data computation that is needed to be performed by PEC (`extern compute()`) in line 18. In addition, PEC needs to consume an additional tick (i.e. `pause` in line 19) before it can start the next rendezvous on the channel CH3. The clock-domain DC continuously waits to receive on channels CH1 and CH2 in synchronous parallel, and moves the diverter to the left or right. The clock-domain TTC similarly waits to receive on channel CH3 and emits a signal, T (line 62), to bring the turntable to the starting position. Finally, O2 is emitted by PEC after the channel communication between the CDs is completed (line 24).

The ICMF unit requires hard real-time guarantees. The conveyor, the diverter, and the turntable operate at different speeds. The conveyor cannot be stopped at run-time due to a backlog of ice creams loaded into the system. There is no way to dynamically vary the conveyor speed, which remains constant throughout the manufacturing process. Given these constraints, it is essential that when the ice cream is detected by the photo-eye and tag reader, the diverter and the turntable are in the correct position by the time the ice cream reaches either one of them. A graphical representation that shows...
// The PEC control logic CD1
{ input int signal A;
  output signal O1;
  output signal O2;
  output channel CH1;
  output channel CH2;
  output channel CH3;
  int signal result op+ = 0;
  while(true){
    LCD1: await(A);
    emit O1;
    // Data computation
    # result = extern process(#A);
    if(#A == 0){
      send CH1;
      // Data computation
      # result = extern compute();
      pause;
      send CH3;
    }
    else send CH2;
    pause;
    emit O2;
  }
} // The DC control logic CD2
{ output int signal D op+ = 0;
  input channel CH1;
  input channel CH2;
  while(true){
    pause;
    // receive on CH1
    LCD2: receive CH1;
    // alternately change direction
    // of diverter
    if(#D==0) #D = 1; else #D = 0;
    emit D;
    E1:pause;
  } ||
  // receive on CH2
  LCD2: receive CH2;
  // alternately change direction
  // of diverter
  if(#D==0) #D = 1; else #D = 0;
  emit D;
  E2:pause;
} // The TTC control logic CD3
{ output signal T;
  input channel CH3;
  while(true) {
    pause;
    // receive on CH3
    LCD3: receive CH3;
    emit T;
    E3:pause;
  }
}

Figure 5.4: Updated SC-SystemJ implementation of ICMF for response time analysis

Figure 5.5: SC-SystemJ control logic controlling ICMF

the possible responses of the system for an incoming ice cream (signal A) is shown in Figure 5.5, where clock-domains CD1, CD2, and CD3 are denoted as PEC, DC and TTC, respectively. There are four possible output events that this system can generate in response to the incoming ice cream detected by the photo-eye (red arrows in Figure 5.5):
5.2 Motivating example

(a) Input and output events based on logical ticks

(b) Reaction time for an input signal A

Figure 5.6: An execution trace of CD1

1. Emission of the signal O1, which informs the monitoring computer that there is an incoming ice cream entering into this manufacturing unit.

2. Emission of the signal O2, which indicates completion of the message delivery from PEC to TTC, or from PEC to DC.

3. Emission of the signal D to trigger the diverter.

4. Emission of the signal T to rotate the turntable

Figure 5.6a shows an execution trace of CD1 from Figure 5.5. CD1 has one input signal A, emitted when an incoming ice cream on the conveyor is detected via the photo-eye and tag reader, and two output signals O1 and O2 to a computer that centrally monitors the overall process. At the beginning of each tick (BOT), all input events (signals) from the environment are captured. The clock-domain then performs computations based on these input events. Lastly, at the end of each tick (EOT), all output signals emitted during the computation are made available to the environment. For example, the trace in Figure 5.6a shows detection of an ice cream (input signal A) at ticks 1 and 6, which also triggers the emission of output signal O1. On the other hand, the time for the emission of O2, in ticks of CD1, varies depending on the rendezvous via channels CH1, CH2 and CH3 between reactions in the clock-domains CD1 and CD2, and CD1 and CD3, respectively.

In the next section, an overview of the real-time analysis of a purely synchronous program is given. Then, in the following section, it is shown that this classical approach is not sufficient to find response times of a GALS program.
5.2.1 Reaction time

The perfect synchrony hypothesis imposes a constraint on each clock-domain that the real clock-domain tick (execution) time should always be shorter than the minimum inter-arrival time of events from the environment such that it will never miss any incoming input events. However, each logical tick takes some finite real time for processing input events, and the reaction time \( \Delta = EOT - BOT \) varies for different logical ticks. Figure 5.6b shows reaction times of CD1 for processing input signal A, where the duration of each logical tick (corresponding to the ticks in Figure 5.6a) is indicated by a rectangular box. For example, CD1 captures an event on signal A at the beginning of tick 1 (at 0 ms), executes the tick for 6 ms, and finally emits O1 to the environment, marking the end of the tick. In this case, the reaction time of the clock-domain was 6 ms. Signal capture and emission can happen at the same tick boundary, e.g. capturing A and emitting O2 at 13 ms, which is both the beginning of tick 6 and the end of tick 5.

In order to satisfy the implicit restriction imposed by the synchrony hypothesis – no input event from the environment can be missed – one needs to calculate the Worst Case Reaction Time \( WCRT = \max(\Delta_i) \), amongst all the possible ticks \( i \) of the clock-domain. The WCRT value must be less than or equal to the minimum inter-arrival time of any two consecutive input signal events from the environment. For example, CD1 in Figure 5.6a has WCRT of 8 ms, which should be less than the inter-arrival time of input events of signal A. It should be noted that WCRT analysis mainly focuses on how fast a program can capture input events [45]. WCRT analysis alone does not determine the maximum time required from input events to generation of all dependent output events as responses to these inputs.

5.2.2 Response time

Response time analysis focuses on the timing relationship between input and output events. However, unlike reaction time, the program response to an input event may take several ticks before generating the corresponding output event(s). For example, emission of the signal O2 in Figure 5.6b, which is triggered by A, may take multiple ticks depending on computation and rendezvous transitions. It is impossible to determine response time from arrival of signal A to emission of signal O2 by only knowing the WCRT of the clock-domains, and without knowing the times spent on communication via channels. This makes response time analysis challenging because it includes ticks and communication via channels from two or more clock-domains. Moreover, emissions of output signals could further be delayed by additional ticks in the presence of signal dependencies within the clock-domain, in accordance with the newly introduced delayed signal communication.
Figure 5.7 illustrates response times for two signals D and T in response to the presence of signal A. As CD1 captures signal A, it tries to send this information to both CD2 and CD3 through channels. However, SC-SystemJ rendezvous over channels requires more than a single tick to complete. In fact, the channel communication is dependent on the speed of both the sender and the receiver clock-domains and the actual method used to exchange data over channels. A message cannot be sent if both the receiver clock-domain and the sender clock-domain are not ready to exchange the message. Moreover, depending on the amount of computation that the clock-domains are required to perform, response times starting from the same input signal may vary from one tick instance to another. In Figure 5.7, two consecutive emissions of signal D upon capturing signal A in CD1 took 9 ms and 15.5 ms respectively, while it took 11.5 ms and 16.5 ms to achieve the same for signal T.

The work described in this chapter focuses on generating a schedule for meeting response time constraints. In order to achieve this goal, one needs to: (1) first verify that the program is functionally correct such that there is always a path in the program which generates output event(s) in response to input event(s), (2) find out the worst case reaction times (WCRT) for each clock-domain when executing on the target processor(s), and (3) schedule the clock-domains on one or more processors.

A number of low-level reaction time analysis techniques already exist \[45, 83\]. Note that these techniques have already studied incorporating communicating latencies e.g., shared memory via time division multiplexing \[91\], and hence, the reaction time includes these latencies. A real-time analysable execution platform, which guarantees bounded execution times for native instructions by removing the source of uncertainties (such as data caches and branch predictions etc.) is a necessity for calculating reaction times. A number of such real-time analysable platforms are available for SystemJ (and also for SC-SystemJ) such as those described in \[92, 93\], whose architectures are inspired from the Tandem Virtual Machine (TVM) \[47\] execution platform. It is assumed that

![Figure 5.7: Response times for A to D and A to T](image-url)
the allocation of clock-domains onto different target processors is given by the system
designer or determined by using some other method, e.g. design space exploration or
multi-objective search [94]. Thus, the main focus of this work is scheduling clock-domains
for guaranteeing response time constraints.

5.3 Specifying real-time bounds

The system designer specifies a set of response time constraints on the SC-SystemJ pro-
gram locations and signals. A system designer may label any pause statement, which
indicates a program location that participates in the response time constraint specifica-
tion, and then use these labels along with input and output signals to specify the parts (or
whole) of the program that need to satisfy different response-time bounds. For example,
certain response time guarantees from CD1 to CD2 and CD1 to CD3 can be specified in
the proposed formulation as shown in Figure 5.8.

Constraints from (5.1) to (5.8) use the timed LTL notation from [95] to specify a set
of bounded response time constraints. The first two constraints (5.1) and (5.2) state that
always (G), when the program is in state LCD1 and input signal A arrives, eventually
(F) signals D and T will be emitted within M and N units of time, respectively. Similarly,
the next two constraints (5.3) and (5.4) also check for the arrival of the signal A but they
do not limit the program to a specific program location. In this case, the constraints state
that always, when the signal A arrives from the environment, and when the program is
in any of the states that can react to the presence of signal A, eventually signals D and
T will be emitted within M and N units of time, respectively. For example, as shown
in Figure 5.4, the presence of signal A is only reacted upon when the program is in the
state LCD1, i.e. await(A). Therefore, although the constraints (5.3) and (5.4) require
the response time bound starting from more than one program location to be satisfied, it
is implicitly restricted to a single location LCD1:pause for this particular example. The
constraints (5.5) and (5.6) check for the response time bound from the program location
LCD1 to LCD2 and LCD3. Lastly, the constraints (5.7) and (5.8) state that when the
signal A arrives, the program will always eventually move to the program locations LCD2

\[
\begin{align*}
G((LCD1 \land A) \rightarrow F_M D) \quad (5.1) & \quad G((LCD1 \land A) \rightarrow F_NT) \quad (5.2) \\
G(A \rightarrow F_M D) \quad (5.3) & \quad G(A \rightarrow F_NT) \quad (5.4) \\
G(LCD1 \rightarrow F_M LCD2) \quad (5.5) & \quad G(LCD1 \rightarrow F_N LCD3) \quad (5.6) \\
G(A \rightarrow F_M LCD2) \quad (5.7) & \quad G(A \rightarrow F_N LCD3) \quad (5.8)
\end{align*}
\]

Figure 5.8: An example of response time constraints
and LCD3 within \( M \) and \( N \) units of time, respectively.

## 5.4 The tool-chain flow for guaranteed response times

The extended tool-chain flow that includes the response time analysis is shown in Figure 5.9, which is extended from Figure 4.1. The approach consists of several steps. The inputs to the design flow are: (1) a SC-SystemJ program, (2) a plant model, (3) the set of response-time constraints, and (4) allocation constraints. The resultant output is the schedule with release times for each program transition that guarantees the response times for the selected execution platform.

In Step-1, the user may annotate the source program locations with labels (e.g. label LCD1 in Figure 5.4) and specify the response time constraints that need to be guaranteed between these program locations, or if no program locations are specified, the response time constraints can be specified with just signals. Optionally, the user may combine the SC-SystemJ program with the plant model, which results in a closed loop control system similar to the approach introduced in Chapter 4. In this particular example, the plant model consists of the turntable, the diverter, and the photo-eye and tag reader from Figure 5.3. These separate components are modelled as SC-SystemJ reactions and combined in synchronous parallel (|) with their individual controller clock-domains. In Step-2, the functional verification is performed (Chapter 4) to check the logical correctness of the SC-SystemJ program. In Step-3, if the user chose to perform response time analysis, the transition times (similar to reaction times) for individual clock-domains are calculated using the techniques described in [82]. In Step-4, the network of FSMS (\( CF_n \) in Figure 5.9) is further transformed into a network of labelled directed graphs (\( G_n \) in Figure 5.9), which labels transitions of the original FSMS with the transition times. Next, an SMT formulation in quantifier free linear real arithmetic (QF_LRA) logic is derived from the network of generated graphs. Furthermore, two additional constraints are formulated and presented to the SMT solver: allocation and response time constraints. Allocation constraints make sure that only one clock-domain can advance to the next state at a time provided there are multiple clock-domains that need to be executed on the same processor. As shown in Figure 5.9 the allocation of every clock-domain in the GALS system to a specific processor may be obtained as a user input or using design space exploration. This guarantees exclusive access to the shared resource. The gamma function \( \Gamma \) (Figure 5.9) rewrites the programmer specified real-time constraints described in Section 5.3 into terms suitable for an SMT formulation.

The resultant SMT formulation is then used as an input to an SMT solver. If the resultant formulation is satisfied, release-times for each outgoing state transition in ev-
The result is an asynchronous interleaved execution of the transitions of individual clock-domains are obtained from the solver; otherwise, a proof of unsatisfiability is obtained. In the rest of this chapter all these steps are described in detail. Once these release times are obtained, the original clock-domain FSMs are executed with a tailored scheduler, which is generated from the solution of the SMT solver obtained in Step-5.
Abstracted FSM

The state transition is labelled as $5.10c$, respectively. Some parts of these FSMs are abstracted for the sake of readability. The state transition is labelled as $5.10c$, respectively. Some parts of these FSMs are abstracted for the sake of readability.

The FSMs for CD1, CD2, and CD3 from Figure 5.4 are shown in Figures 5.10a, 5.10b, and 5.10c, respectively. Some parts of these FSMs are abstracted for the sake of readability. The state transition is labelled as $Boolean guard on Input signals$ $Output/Internal Actions$ $For example, the PEC clock-

domains, unlike multi-rate programs where all clocks are derived from a single super-fast clock $[96]$.}

5.4.1 Compiling the motivating example into a network of FSMs

The FSMs for CD1, CD2, and CD3 from Figure 5.4 are shown in Figures 5.10a, 5.10b, and 5.10c, respectively. Some parts of these FSMs are abstracted for the sake of readability. The state transition is labelled as $Boolean guard on Input signals$ $Output/Internal Actions$ $For example, the PEC clock-
domain (CD1) can make the transition from \(LCD1\) to \(N28\) and perform actions (by emitting the signal \(O1\) and executing \(\text{process}(\#A)\)) when the guard \(A \land (\#A = 0)\) is evaluated to true. The dotted circles indicate rendezvous states – states that participate in channel communication. Each transition in the FSM represents the big-step from one tick to the next (Chapter 3); multiple transitions from different synchronous parallel reactions might be encapsulated in a single transition. For example, every outgoing transition from state \(N177\) (Figure 5.10b) indicates transitions of the individual reactions composed together in parallel via the statement \{receive CH1; \ldots\} \parallel \{receive CH2; \ldots\} in Figure 5.4. CD1 (Figure 5.10a) starts from the start state \(st1\) and unconditionally moves on to the state labelled \(LCD1\) (\(\text{await}(A)\) statement in Figure 5.4). Three possible transitions can now be performed depending on the presence and value of signal A. For example, nodes \(N28\) and \(N51\) represent the if and else branches in CD1 (lines 15 and 22 in Figure 5.4), respectively. On the other hand, the FSM structure of CD2 is more complex due to the synchronous composition of the channel communications. For example, a state transition of CD2 when it is in one of the rendezvous states such as \(N177, N108\) and \(N146\), is dependent upon the status of both channels CH1 and CH2. Upon completion of the rendezvous, signal D is emitted, and the correct data computation is performed. After starting from \(st3\) (Figure 5.10c) CD3 unconditionally moves to state \(N14\), an internal rendezvous state, and remains in that state until the rendezvous with CD1 is complete. Upon completion of rendezvous, signal T is emitted.

The figures only show some of the guards and outputs to avoid clutter. Furthermore, the channels CH1, CH2, and CH3 have been replaced with signals \(CH1_{req}/CH1_{ack}\), \(CH2_{req}/CH2_{ack}\) and so on, which are used to implement rendezvous by performing a four-phase handshake. This approach of implementing rendezvous using signals rather than channels, like in ordinary CSP [26], is essential to guarantee response times. In the upcoming sections, these essentials will be further elaborated.

### 5.4.2 Formalising programmer specified real-time constraints

Once the compiler generates a network of FSMs, programmer specified real-time bounds (such as those in Figure 5.8) are further transformed into terms suitable for the SMT formulation. Let \(C_t\) represent the set of all programmer specified real-time constraints. Then a function \((\Gamma)\) maps each constraint to a set of the three tuples \(\{(v_i, v_j, N^{ij}), \ldots\}\) where element \(v_i\) is a starting state in the network of FSMs, \(v_j\) is a destination state in the network of FSMs and the third element \(N^{ij}\) is the required real-time bound from \(v_i\) to \(v_j\). For example, constraint (5.1) in Section 5.3 is mapped to the set \(\{LCD1,N108,M\}, \{LCD1,N146,M\}, \ldots\}\), see Figures 5.10a and 5.10b. For constraint (5.5), the channel statements receive CH1 and receive CH2, annotated with LCD2, in
the DC clock-domain (Figure 5.4) are equivalent to all dotted states in Figure 5.10b. Hence, the resultant set is \{ \((LCD1, N177, M)\), \((LCD1, N108, M)\) \ldots \}. On the other hand, the sets mapped from constraints (5.2) and (5.4) contain only a single tuple, which is \{ \((LCD1, N21, N)\) \}. Finally, all the tuples in the constraint sets are combined with logical conjunction (\(\wedge\)) to give the final set of constraints on FSM states that needs to be satisfied by the generated schedule.

### 5.4.3 Reaction time of a clock-domain in a GALS program

As already explained previously in Section 5.2.2, finding reaction times for each clock-domain is one of the requirements in the response time analysis of a GALS program. The reaction time in pure synchronous programs (à la Esterel) is well known – it is the duration of an iteration of the synchronous program, from the start of an iteration (beginning of the tick) to the start of the next iteration (beginning of the next tick, which is equal to the end of the current tick) [56]. In a GALS setting the reaction time is not obvious due to communication between clock-domains. Consider blocking rendezvous like in CSP: a clock-domain halts (stops processing) when in the send (e.g. send CH1 in Figure 5.4) or receive (e.g. receive CH1 in Figure 5.4) state. Consequently, no input event can be captured even by the other reactions in that clock-domain. A chain of such dependencies can easily be created in any GALS/asynchronous program. In order to compartmentalise the reaction time to a single clock-domain, it is essential a clock-domain does not halt. A solution to this problem is blocking at the logical level rather than halting (Section 3.5.2), which rewrites all rendezvous communication as synchronous programs using signals. For example, the channel CH1 and its corresponding send (send CH1) and receive (receive CH1) statements in Figure 5.4 are rewritten into algorithms performing the 4-phase handshake using signals \(CH1_{req}\) and \(CH1_{ack}\). Upon this rewrite, communicating clock-domains act as an environment to each other, and well known reaction (equivalently, transition) time computation techniques [17, 82] can be used to calculate the reaction time of each clock-domain individually.

![Figure 5.11: Adding transition times to the edges](image-url)
In order to enable response time analysis, individual state transitions in the FSMs, generated from the process described in Section 5.4.1, are annotated with transition times. As a result, a new labelled graph is created. Figure 5.11 shows an example of the labelled graph for CD3 (TTC) from the motivating example. As one can see, each edge in the graph is labelled with the transition time $w_{i,j}$. The time varies for each transition depending on the amount of computation. This information is important for guaranteeing the response times of the SC-SystemJ programs because, depending on the path taken, the time required to reach a certain state can vary.

The next section presents the formulation of the response time problem using SMT. First, the solution for a single processor system is described. Then, this constraint is relaxed for shared/distributed memory multi-processor systems.

5.4.4 SMT formulation for a single processor system – scheduling a single FSM

First, consider the case of scheduling a single FSM by converting it into an edge-labelled graph with the edges annotated with the transition time. Note that transforming FSMs into edge-labelled graph, or vice-versa, is trivial since all states and state transitions from one state to another in the FSM can be directly mapped to vertices and edges in the edge-labelled graph. Therefore, the terms state and vertices, and edges and transitions are used interchangeably from this point onwards. Every graph is a tuple $(V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Each edge $e = (b(i), O, w_{i,j}^k, v_i, v_j)$, $\forall e \in E$, is labelled with: the Boolean function $b(i), \exists i \in I$, where $I$ is the input signal set, the output action set $O$, the edge transition time $w_{i,j}^k \in W$ and the source and destination vertices $v_i \in V, v_j \in V$, respectively. The annotation $k$ (in $w_{i,j}^k$) is used to distinguish multiple edges between the vertices $v_i$ and $v_j$. Let $V_s \subseteq V$ be the set of source vertices of the graph (vertices without incoming edges, i.e., $\deg^{-}(v) = 0, \forall v \in V_s$). Let $C_t$ be the set of real-time constraints specified by the designer, as explained in Section 5.4.2. Recall that in order to incorporate the constraints given by a designer in the SMT formulation, a gamma function is used $\Gamma : C_t \rightarrow \{(v_i, v_j, N^{ij}), \ldots \}, v_i \in V, v_j \in V^{ij}, N \in \mathbb{R}^{\geq 0}$, which maps each constraint to a set of three tuples, where $N^{ij}$ is the required real-time bound for transition from $v_i$ to $v_j$. Finally for some $v_i \in V$, let $r_i$ denote the release-time for the outgoing edge(s) for the vertex $v_i$. Note that all outgoing edges of $v_i$ have the same release time, hence $r_i$ is not qualified with “$k$”. Then, the response time satisfaction problem can be formulated in QF_LRA as:
The tool-chain flow for guaranteed response times

\[ \bigwedge_{v_i \in V_s} r_s \geq 0 \quad \land \quad \bigwedge_{v(b(i),O,w_{i,j},v_{i,j}) \in E} r_j \geq r_i + w_{i,j} \land \]
\[ \bigwedge_{v(t) \in C_t} \bigwedge_{\forall(v_i,v_j,N^{ij}) \in \Gamma(t)} r_j + \chi(v_j,B) - r_i \geq N_{i,j} \]

Equation 5.9 states that the outgoing source vertex transitions can start at any time greater or equal to zero. The successors get delayed by the transition time due to the dependency from their predecessors and the constraints specified by the designer need to be satisfied. Function \( \chi : v_j \rightarrow B \rightarrow \mathbb{R}^{\geq 0} \) maps a destination state \( v_j \) and the boolean expression \( B \) to a real number greater than or equal to zero, which is a worst-case transition time from \( v_j \) to its immediate successors that satisfy \( B \). This function is required since the final output response is generated at the end of state transition (i.e. clock-domain tick), and in the worst case it is the longest transition time among all possible out-going edges from \( v_j \). Note that the release time of the outgoing edges of join vertices – vertices with multiple incoming edges (e.g., \( N_{146} \) in Figure 5.10b) get delayed by the worst incoming edge transition time.

5.4.5 SMT formulation for a single processor system – scheduling multiple FSMs

Let \( F = \{(V_0, E_0), \ldots, (V_{n-1}, E_{n-1})\} \) denote the set of edge-labelled graphs for \( n \) CDs running on the same processor. Then, one needs to schedule the execution of these CDs with exclusive access to the underlying processor. First, the definition of a new function \( \Omega : v_i \rightarrow \mathbb{R}^{\geq 0}, \forall v_i \in V_j, \forall (V_j, E_j) \in F \) is given as follows:

\[ \Omega(v_i) = \max(w_{i,j}^k), \quad \forall e_i \in E_i, E_i \subseteq E_j \]

Function \( \Omega \) gives the worst transition time amongst all the outgoing edges (set \( E_i \)) of any vertex \( v_i \) in the graph. Then, the exclusive access to the underlying processor can be expressed as:

\[ \bigwedge_{\forall (V_i, E_i) \in F} \bigwedge_{\forall (V_j, E_j) \in F, j \neq i} \bigwedge_{\forall v_x \in V_i, \forall v_y \in V_j} r_x \geq r_y + \Omega(v_y) \lor r_y \geq r_x + \Omega(v_x) \] (5.10)

A conjunction (\( \land \)) of Equation 5.9 and Equation 5.10 is performed to obtain the schedule. The above formulation can schedule a network of the original FSMs without rendezvous communication on a single processor. More than one outgoing edge from any vertex \( v_i \) can be created due to open inputs (i.e. input signals not generated by
the plant model) and the conditional branches evaluating these inputs during program run-time. These branches can have different transition times depending on the program logic. Therefore, using the worst transition time from amongst all the outgoing edges of \(v_x\) and \(v_y\) in Equation 5.10 guarantees exclusive access of FSM transitions on the underlying shared processor. However, the above formulation might still be unable to schedule valid FSMs due to loops (e.g., FSM in Figure 5.10c).

Bounding temporal loops

Strongly connected components in the FSMs (or equivalently, in the edge-labelled graph) occur in two forms: (1) self-loops, which indicate a waiting state, usually for an input signal, e.g., the self-loop on vertex LCD1 in Figure 5.10a which waits for signal A, and (2) back-edges, which indicate explicitly programmed loops, e.g., the loop statement enclosing CD1 (see Figure 5.4 and Figure 5.10a) is indicated with edges back to vertex LCD1 from its successors. Both these loops are temporal, i.e., every edge consumes one logical tick. Temporal loops do not need any special attention in reaction time analysis because reaction time only considers a single reaction (one logical tick) of a CD, but need special consideration in response time analysis. For example, if a designer wants to know the response time from state LCD1 to state N28 (in Figure 5.10a), the answer is simply \(\infty\). Without knowing when input signal A will be present, one can only assume that it might never be present, hence the answer. Bounding loops waiting on input signals from the environment is the primary reason for composing the clock-domains with their respective plant models, otherwise the response can never be guaranteed. In this work it is assumed that the plant model is generated manually by the designer, similar to when verifying liveliness properties (Section 4.2.2). Explicitly programmed loops (e.g. `while(true){...}`) also create back-edges in the generated edge-labelled graph. For example, SC-SystemJ reactions can have a main loop that repeats upon completion. Such loops are removed during the SMT formulation since a schedule synthesised from the SMT solver will be repeated, hence emulating a program loop.

Finally, there are loops that cannot be bounded simply by modelling the plant (e.g., self-loop on vertex N14 in Figure 5.10c), since these loops correspond to signals created for rendezvous communication via channels. The bound of these loops can only be ensured by the scheduling policy of the clock-domains. For example, signal \(CH3_{req}\) is only emitted (if ever) from CD1 to CD3 depending on the scheduling policy of these clock-domains. However, obtaining the scheduling policy is the objective of this work itself! A solution for this paradoxical chicken-egg problem is presented next.
Accounting for rendezvous communication dependencies

Clock-domains are said to have entered rendezvous once the program control-flow encounters a send or receive statement. Clock-domains perform rendezvous via a pair of rendezvous signals and additional states generated from the channel rewrite rules as shown in Algorithm 3.1 and Algorithm 3.2 (ACK and REQ) from Section 3.5.2. To illustrate the four-phase handshake protocol, in a comprehensive way, consider the example between receiving/sending clock-domains communicating through channels (receive CH3 and send CH3) from Figure 5.4 with their rendezvous state representation, and dependencies between those states, shown as dashed edges, in Figure 5.12a. Vertices N399, N59, N14, and N12 in Figure 5.10a and Figure 5.10c are called S1, S2, R1, and R2 in Figure 5.12a, respectively, for sake of explanation.

Vertices S1 and S2 are the rendezvous states from the sender (CD1), generated by the pause statement inserted by the rewrite shown in Algorithm 3.1 and R1 and R2 are from the receiver (CD3) resulted as shown in Algorithm 3.2 respectively. When the sender first enters state S1 it waits for rendezvous signal \( CH3_{\text{ack}} \) from the receiver. On the other hand, the receiver is able to make a transition to R2 from R1 in the absence of \( CH3_{\text{req}} \) from the sender. The sender is then able to make the transition to S2 where it also continuously emits \( CH3_{\text{req}} \) until the absence of the signal \( CH3_{\text{ack}} \). Finally, the receiver first finishes the rendezvous (i.e. \( CH3_{\text{req}} = \top \)) followed by the sender (i.e. \( CH3_{\text{ack}} = \bot \)).

In order to encode channel communication in the SMT formulation, dependency edges between the rendezvous states are inserted. For example, when the sender is in S1 it can only make a transition to S2 after both its predecessor state (Pred1) and outgoing transition from R1 in CD3 complete their execution (i.e. \( CH3_{\text{ack}} = \top \)). On the other hand, R1 is only dependent on its predecessor (Pred2). Dependency from R2 to S2
indicates that the sender is only able to complete rendezvous after scheduling the outgoing transitions of the receiver state $R2$ (i.e. $CH_{3 \text{ack}} = \bot$). Similarly, the receiver is only able to complete rendezvous when the sender enters $S2$ (i.e. $CH_{3 \text{req}} = \top$). A complete encoding of the dependency constraints of the rendezvous states is shown in Figure 5.12b. These rendezvous constraints are automatically encoded via Equation 5.9 because of the added dependency edges.

Finally, the self-loop in each rendezvous state, which indicates waiting on the presence or absence of rendezvous signals, is omitted in the formula. Instead, dependency edges replace self-loops in the SMT formulation.

**Interplay of rendezvous and synchronous parallel reactions**

Although the aforementioned encoding approach allows one to schedule multiple clock-domains with channels, it has one limitation: it cannot correctly handle rendezvous communication over a distinct channel carried out in parallel using the synchronous parallel operator (\(|\|\)), e.g., \(\{\text{receive CH1; } \ldots\}\|\{\text{receive CH2; } \ldots\}\) (in Figure 5.4).

In Figure 5.13a (reproduced from Figure 5.10b), $N146$ is one such state. Two incoming channel dependency edges ($CH_{1 \text{req}}, CH_{2 \text{req}}$) are generated for $N146$ from CD1. There are several possible outgoing transitions from $N146$: (1) only receive CH1 is ready to rendezvous, (2) only receive CH2 is ready to rendezvous, (3) both are ready to rendezvous, and (4) none of them are ready to rendezvous. In Figure 5.13a, only cases (1) and (2) are shown for clarity of explanation. Suppose CD1 enters the rendezvous state of channel CH1 and settles in state $N25$ (Figure 5.10a). CD2 should be able to make a transition to the successor state $N174$ since $CH_{1 \text{req}}$ is set to true by CD1. However, this transition is not possible because of the dependency edge ($CH_{2 \text{req}}$) also coming from CD1 (send CH2). As a result, this macro state representation prevents micro transition of individual synchronous parallel reactions.

To address this problem multiple dependency edges from multiple clock-domains, re-
5.4 The tool-chain flow for guaranteed response times

resulting from synchronously composed rendezvous states, are merged into a single dependency edge where the guards are combined with logical disjunction ($\lor$). The updated representation of $N_{146}$ is shown in Figure 5.13. $N_{146}$ now has only one dependency edge. $N_{146}$ can be scheduled immediately when either of the two guards ($CH_{1\text{req}}, CH_{2\text{req}}$) used for the channel communication are enabled.

Figure 5.14 shows a single execution trace (a run) of the automated ICMF unit controller produced by the SC-SystemJ compiler for a single processor system with the following constraints: $A \rightarrow F_{80,55}D$ and $A \rightarrow F_{70,55}T$. The widths of the boxes indicate the reaction time for each transition in the FSM. For example, the transition from $LCD_{1}$ to $N_{28}$ of $CD_{1}$ in Figure 5.10a is 7 time units (from 37 to 44 in Figure 5.14). The other interesting points are:

- There is no single integral multiple relating the execution speed (clocks) of different clock-domains. Thus, no super-fast clock can be derived like in multi-rate systems. The execution of clock-domains is truly interleaved.

- In the general case, one cannot achieve 100% processor utilisation. In Figure 5.14 the processor is idle from 53 to 63 time units. When multiple clock-domains are allocated to a shared resource, their execution needs to be serialised. This serialisation is achieved using Equation 5.10 where the release time of a transition amongst any clock-domain is delayed by the worst case amongst all the outgoing edges from the current state of the program. For example, release of the outgoing transition of $N_{25}$ in $CD_{1}$ is delayed by $\Omega(N_{146})$ of $CD_{1}$, which is $w_{N_{146},N_{161}} = 18$ time units in Figure 5.10b. The worst case transition time is chosen because one cannot know which transition will be taken in reality.

Figure 5.14: A schedule for ICMF benchmark with response times. All clock-domains are allocated to a single processor
5.4.6 Extension to the multi-processor case

Equation 5.10 assumes exclusive access by states and their transitions in the FSM to the underlying processor. For the multi-processor case, Equation 5.10 needs to be modified. Only those FSMs (and their related states) that are allocated on the same processor should be considered as having exclusive access.

Let $P$ be the set of processors in the system. Then, $\forall p \in P, F_p \subseteq F$ is the set of FSMs on processor $p$. The Equation 5.10 is then modified to:

$$\forall p \in P, \bigwedge_{(V_i,E_i) \in F_p} \bigwedge_{(V_j,E_j) \in F_p,j \neq i} \bigwedge_{v_x \in V_i, v_y \in V_j} r_x \geq r_y + \Omega(v_y) \vee r_y \geq r_x + \Omega(v_x)$$  \hspace{1cm} (5.11)

It is assumed that the transition time analysis (Section 5.4.3) already integrates potential delays which can be introduced by sharing resources (e.g. communication links). This can be done during static WCRT analysis [17, 96]. If all the three clock-domains in Figure 5.4 are allocated on a single processor system, then Equation 5.11 guarantees that the states and their transitions in the three corresponding FSMs have exclusive access to this underlying processor. But, in the case where all three clock-domains are allocated onto different processors, as shown in Figure 5.3, Equation 5.11 results in zero constraints, because $|F_p| = 1, \forall p \in P$.

Figure 5.15 shows the schedule when all clock-domains are allocated to different processors, while satisfying the same response time constraints: $A \rightarrow F_{80.55}D$ and $A \rightarrow F_{70.15}T$ (as for Figure 5.14). Clock-domains are now able to run in parallel on individual processors and their execution is parallelised. In the case of parallel execution the exclusive

![Figure 5.15: A schedule for ICMF benchmark with each of the three clock-domains running on their individual processors](image-url)
access constraint in Equation 5.10 is no longer applicable. Hence, the previously seen delays in program execution (53 to 63 time units in Figure 5.14) do not occur in parallel execution. Instead, there are delays in the individual clock-domain executions due to rendezvous communication, for example, 20 to 28 time units in Figure 5.15. The response time from signal A to T has been reduced, while the response time from A to D remains the same. Schedules shown in Figure 5.14 and Figure 5.15 are the optimal response times obtained by iteratively tightening the solution attained from the SMT solver using the binary search algorithm.

## 5.5 Complexity analysis

The number of generated clauses (constraints) in the previous formulation (Equation 5.9 and Equation 5.12) is polynomial in the number of variables (states in the FSM). The number of constraint clauses for Equation 5.9 is bounded in the worst case as shown in Equation 5.12 where $E$ is the set of edges, and $F$ is the set of FSMs:

$$O(|E| \times |F|)$$

$$\sum \forall_{p \in P} \left( |F_p| \sum_{i=1}^{|F_p|} (|F_p| - i) |V|^2 \right)$$

For Equation 5.11, the complexity is shown in Equation 5.13, which can be simplified as follows:

$$\left( \sum_{i=1}^n i = \frac{n(n+1)}{2} \right) \Rightarrow$$

$$\left( \sum_{i=1}^{|F_p|} (|F_p| - i) = \sum_{i=1}^{|F_p|} |F_p| - \sum_{i=1}^{|F_p|} i \right) \Rightarrow$$

$$\left( |F_p|^2 - \frac{|F_p|(|F_p| + 1)}{2} \right) \Rightarrow$$

$$|F_p| \left( \frac{|F_p| - 1}{2} \right)$$

$$\Rightarrow$$

$$\sum \forall_{p \in P} \left( \sum_{i=1}^{|F_p|} (|F_p| - i) |V|^2 \right) =$$

$$O \left( \sum \forall_{p \in P} \left( |F_p| \left( \frac{|F_p| - 1}{2} \right) |V|^2 \right) \right)$$
### Table 5.1: An overview of the benchmark programs

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Lines of code</th>
<th>Number of reactions</th>
<th>Number of clock-domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICMF</td>
<td>65</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Conveyor+Arm</td>
<td>39</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Robot</td>
<td>32</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Pacemaker</td>
<td>108</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table 5.2: Benchmark results

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Specified response time constraints</th>
<th>Response time bounds (with varying # of processors)(^a)(^b)</th>
<th>Time interval between input and output signals (with varying # of processors)(^b)</th>
<th># of FSM states in SMT formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>ICMF</td>
<td>(G(A \rightarrow F_{80.55}D))</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>ICMF</td>
<td>(G(A \rightarrow F_{70.15}T))</td>
<td>48</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>Conveyor+Arm</td>
<td>(G(A \rightarrow F_{55.3}Move))</td>
<td>31</td>
<td>31</td>
<td>–</td>
</tr>
<tr>
<td>Robot</td>
<td>(G(A \rightarrow F_{72.5}Picked))</td>
<td>49</td>
<td>47</td>
<td>–</td>
</tr>
<tr>
<td>Pacemaker</td>
<td>(G(A \rightarrow F_{190}Pulse))</td>
<td>105</td>
<td>76</td>
<td>–</td>
</tr>
</tbody>
</table>

\(^a\) Units are in milliseconds

\(^b\) Time intervals = \([t_s, t_e]\) where \(t_s\) ≡ start time for input signal and \(t_e\) ≡ end time for output signal

Response time bounds = \(t_e - t_s\)

The final and clause in Equation 5.11 gives the \(|V|^2\) term in Equation 5.13. The first two and clause give the \(\sum_{i=1}^{\lvert F_p \rvert} (\lvert F_p \rvert - i)\) term. The complexity is dominated by the \(|V|^2\) term when many (or all) FSMs are allocated to a single processor. The number of generated clauses reduces with increasing number of processors. Since the SMT solve times increase exponentially with growing number of clauses, it is expected that the solve times would decrease with increasing number of processors, which is in fact shown in the experimental results presented in the next section.

### 5.6 Benchmark results

A number of experiments were carried out to obtain schedules that satisfy response time requirements for various benchmark programs. [Table 5.1](#) gives an overview of the complexity of the benchmark programs, which shows the lines of SC-SystemJ source code and total numbers of both reactions and clock-domains in each program. ICMF is the ice cream manufacturing facility case-study described in this chapter. For descriptions of all other examples the reader is referred to [Section 4.3](#) as they are reused in this experiment. [Table 5.2](#) shows the response time requirements for ICMF as the times between the arrival of an ice cream (signal A) and the emissions of two output signals D and T, which

---

\(^1\) This is a standard series for finding a triangular number 97
5.6 Benchmark results

Figure 5.16: Z3 SMT solver runtime with varying # of constraint clauses

control the diverter and turntable. The time must always be less than 80.55 and 70.15 time units (i.e. second and third rows of the second column), respectively. For the pacemaker, on the other hand, it is checked whether electrical pulse is generated within a given time constraint if the heartbeat is not detected within a set time period. In the conveyor and mechanical arm example, the response time for the system to generate a control signal for moving the conveyor once an item is placed on a conveyor belt has been verified. In the robot motion controller, once the controller detects an item, the time required to generate a control signal after picking up the item is bounded. The response time bounds for these benchmark programs are written in the form of $\text{Equation 5.3} (G(A \rightarrow F_{\text{time}}S))$ as shown in the second column of Table 5.2, where $A$ is an input event, $S$ is an output event, and $\text{time}$ is the required time bound between the two events.

The result of the response time analysis is also shown in Table 5.2. As one can see from the third to the fifth columns in the table, the SMT solver gave reduced response time bounds for the benchmark programs Robot, Pacemaker and ICMF $G(A \rightarrow F_{70.15}T)$ with increasing number of processors. The reduction is due to the fact that multiple clock-domains can be executed in parallel on a multi-processor system. Some examples such as ICMF $(G(A \rightarrow F_{80.55}D))$ and Conveyor+Arm show no improvements with increasing number of processors. This is due to the fact that the computations (i.e. ticks) required for generating these outputs could not be parallelised. Furthermore, the release times for the state transitions have decreased with increasing number of processors for all examples (i.e. start and end time for the response time bounds) thanks to the parallel execution of clock-domains. Figure 5.16 shows run-time values for the SMT solver to obtain the optimal (via binary search) and satisfactory response time bounds, respectively. The time required to obtain an optimal solution is on average 7 times longer than the time required to obtain a satisfactory solution.

We also increased the complexity of two of our benchmark example, ICMF and Robot, in order to investigate the scalability of the approach for larger systems. In this experiment, only the runtime for finding satisfactory solutions is shown, since that is the goal
of this work. The number of controllers used in the ICMF and the Robot system have been increased by up to 5 and 4 times, respectively, leading to a maximum of 16 processor utilisation. All FSMs (clock-domains) are evenly distributed across the processors in order to effectively utilise available resources. Figure 5.17a and Figure 5.17b show the results of the experiments. The SMT solver took the longest in all cases with a single processor system was 575 seconds, which is approximately 500 times larger than the two processor system. Solving the single processor problem for Robot example also
5.6 Benchmark results

took the longest time. Except for the first two largest single processor systems, the SMT solver took less than 2 seconds to solve all other configurations. Lastly, Figure 5.17c shows the number of constraints generated by the compiler for 1 to 4 processors. For all cases, the single processor systems have the largest number of constraints (e.g. 9750 for 5 ICMF units and 7784 for 8 Robot systems). The number of constraints shown in Figure 5.17c includes the results of both Equations (5.9) and (5.14), which are the dependence and resource constraints, respectively. The benchmarks have shown that the proposed technique can solve the guaranteed response time analysis problem modelled as a satisfactory problem in a reasonably scalable manner, especially with increasing number of processors. On the other hand, the solver runtime increases significantly when obtaining the optimal solution, which is expected since finding an optimal solution to the scheduling problem is known to be NP-hard [98].

To conclude, this chapter has described a novel technique for analysis and scheduling of GALS programs to guarantee bounded response times. A precise definition of response time in a GALS setting has been given, and then a Satisfiability Modulo Theory (SMT) based formulation in the quantifier free linear real arithmetic logic is performed to tailor a scheduler for the GALS program that meets the response time constraints. This chapter proposed novel techniques to deal with challenges such as reaction time analysis, and the interplay of synchronous parallel execution with rendezvous communication between clock-domains. The benchmark results are very encouraging and show that the SMT based approach is scalable, especially as the number of processors in the design system increases.
SC-SystemJ and Esterel are a class of programming languages used for designing and implementing systems that continuously respond to input from their environment. These languages have been successfully used in programming a plethora of systems such as fly-by-wire in Airbus [99], security surveillance systems [48], a model of an implantable pacemaker for the heart [32], etc. Usually, these systems also need to meet real-time constraints imposed by the environment. The previous chapter introduced a technique to guarantee response times of GALS systems designed in the SC-SystemJ language. Yet, SC-SystemJ and other reactive languages do not support the description of real-time statements as first class language constructs. For example, one cannot describe a simple real-time delay (postponement) of an operation for 0.2 ms. Although these languages are based on formal semantics, which makes them good for verifying functionally correct programs, they leave guaranteeing non-functional properties such as timing behaviour as an implementation detail [17]. This is because physical time cannot be incorporated without information about the underlying execution platform. Although there exist many static analysis techniques [17] to ensure that a designed system meets real-time constraints, these languages do not provide the designer an explicit control of the system’s timing behaviour. Moreover, the real-time period of the discrete logical tick in reactive languages is elastic, which makes it more difficult for designers to utilise physical time in the program.
This chapter targets enhancement of the expressiveness of formal GALS and synchronous languages by adding three real-time constructs: `wait_inbetween`, `wait_atleast`, and `wait_exact`. These real-time constructs speed up the design process by allowing system designers to explicitly control the real-time behaviour of the program using the SC-SystemJ language itself and without reliance on external resources such as timers, which can induce unwanted program behaviours. This will be explained in Section 6.6, illustrating when timers are used incorrectly with synchronous or GALS languages. The introduction of real-time as a language construct allows verification of temporal properties of programs before deployment.

### 6.1 Motivation via examples of real-time systems programmed in SC-SystemJ

Real-time operating systems usually provide two types of mechanisms for introducing real-time constructs in the developed program: (1) the ability to perform a timeout and (2) the ability to run tasks periodically with some period $T$. The work presented in this chapter provides similar mechanisms in reactive languages for real-time program development. In fact, this approach provides exact and non-exact real-time control mechanisms – similar to the two real-time mechanisms present in real-time operating systems.

#### 6.1.1 Programming using the non-exact real-time control construct: the `wait_inbetween` statement

We introduce the `wait_inbetween(M..N)` statement which postpones the program control flow from proceeding to the following statement for a minimum of $M$ time units and a maximum of $N$ time units, as shown in Figure 6.1b.

We demonstrate the usefulness of the `wait_inbetween` construct using a system that measures human response time. Consider a machine used to test human response time which collects data for future research purposes. The machine switches on a green light on a touch screen for anywhere in between 50.3 ms to 200.3 ms; if the user can touch this green light then she is successful and an integer 1 is sent to another process in order to increase an integer counter called `successful`. If unsuccessful a 0 is sent instead to increase a counter called `unsuccessful`. This data can then be used for further statistical analysis on human responsiveness. The SC-SystemJ code for such a system, along with the expected timing behaviour of the `wait_inbetween` statement, is shown in Figure 6.1. The SC-
6.1 Motivation via examples of real-time systems programmed in SC-SystemJ

```java
{ // Clock-domain 1
    input signal TOUCH;
    output signal GREEN_LIGHT;
    output int channel S;
    while(true) {
        signal T;
        abort(T){
            // Abort if user touches the screen
            abort(TOUCH);
            {sustain GREEN_LIGHT; //reaction R1
                || // synchronous parallel operator
                { //exit after any where from 50.3 to 200.3 ms.
                    wait_inbetween (50.3 .. 200.3 ms);
                    emit T;
                    pause;
                }
            // reaction R2
        }
        present(TOUCH) send S(1); // Send 1 if touched
        else present(T) send S(0); // Else Send 0
        // pass some time before restarting
        pause; pause; pause;
    }
}

{ // Clock-domain 2
    input int channel S;
    int signal successful op+ = 0;
    int signal unsuccessful op+ = 0;
    while(true){
        receive S;
        if(#S == 1)
            #successful = #successful + 1;
        else
            #unsuccessful = #unsuccessful - 1;
    }
}
```

(a) A human responsiveness system (HRTCS)

(b) Timing diagram – the X-axis shows the execution time of the The vertical arrow shows one example of the expiration of the `wait_inbetween(50.3 .. 200.3 ms)` statement.

Figure 6.1: Programming a human responsiveness system (HRTCS) with `wait_inbetween`
abort(T) {
  abort(DoorOpened) {
    wait_atleast(10000 ms);
    emit T;
    pause;
  }
}

(a) Timeout waiting for the input signal DoorOpened

(b) Timing diagram – the X-axis shows the execution time of the wait_atleast statement. The vertical arrow shows the possible expiration of wait_atleast (10000 ms) statement

Figure 6.2: Programming a timeout with wait_atleast

SystemJ program in [Figure 6.1a] is divided into two clock-domains, the first clock-domain continuously displays a green light (GREEN_LIGHT signal) on the touch screen sensor and waits for the user to respond. If the user is able to touch the screen within the specified time of at least 50.3 ms and at most 200.3 ms, a positive response (i.e. an integer number 1) is sent to the second clock-domain. Most of the SC-SystemJ programming constructs used to implement this system are already described in Chapter 3. The wait_inbetween statement, however, which models the non-exact postponement between 50.3 ms and 200.3 ms is missing in the SC-SystemJ language presented in Chapter 3. Such non exact real-time mechanisms are useful for developing systems where the real-time postponement is not known or should not be known a priori.

6.1.2 Programming using the non-exact real-time control construct: the wait_atleast statement

The wait_atleast (M) statement is used to postpone the program control-flow from proceeding to the next statement for a minimum M time units. The maximum postponement is unbounded, but countable.

There are many instances when one would like to wait on an input from the environment for only a specified amount of time. If the signal is not received, then a timeout is generated so that the program can make progress. Such a timeout programmed in the SC-SystemJ language, along with its timing behaviour is shown in Figure 6.2. The SC-SystemJ program waits for 10 seconds on an input signal DoorOpened from the envi-
6.1 Motivation via examples of real-time systems programmed in SC-SystemJ

```java
while(true) {
    wait_exact (1 ms);
    emit S;
    //do something
}
```

(a) Periodically emitting signal S

![Timing diagram](image)

(b) Timing diagram – the X-axis shows the execution time of the `wait_exact` statement. The vertical arrow shows the only possible expiration of the `wait_exact (1 ms)` statement.

Figure 6.3: Programming a periodic task with `wait_exact`

...environment, and makes progress even if the signal is not received after 10 seconds.

The timing diagram, [Figure 6.2b](image), shows that the `wait_atleast` statement waits for at least 10 seconds, but unlike `wait_inbetween` there is no upper bound. For example in [Figure 6.2b](image), the program might proceed to the next statement at anytime greater than or equal to 10 seconds (say 11 seconds) according to the semantics of `wait_atleast` statement. The statement following the `wait_atleast` statement is executed after a finite countable, but unbounded postponement. Such a statement is useful when a clock-domain or a reaction needs to suspend execution without a strict real-time requirement.

### 6.1.3 Programming periodic tasks using the exact real-time construct: the `wait_exact` statement

The previous `wait` statements are non exact, i.e. the time of execution of the statement following the `wait` cannot be controlled exactly. The final variant of the `wait` statement allows doing exactly that; `wait_exact` can be used to program exact timeouts, periodic tasks, etc.

One such example: the periodic emission of a signal to the environment is shown in [Figure 6.3](image). As before, upon execution of the `wait_exact (1 ms)` statement, time starts elapsing. But unlike in previous cases the following statement needs to be executed exactly after 1 ms has passed, as shown in [Figure 6.3b](image).

The major motivations for introducing these real-time `wait` mechanisms, and at the same time, the contributions of the work presented in this chapter are: (1) real-time waits do not affect the model of logical time in SC-SystemJ programs as long as the amount...
of delay is within boundaries that can be determined statically by program analysis; (2) the waiting model relies on the use of relative instead of absolute real-time, i.e. a wait in selected time units is counted from the beginning of the wait statement; and (3) these waiting mechanisms allow mixing of behaviours with real-time features and others with only logical time. All these as well as semantics of the wait constructs, are discussed and illustrated in the rest of this chapter. These contributions can be further refined as follows:

1) Real-time is specified in the $\mathbb{Q}^{>0}$ domain of numbers, i.e. non-negative rational numbers.

2) The work presented in this chapter is the first, to our knowledge, to allow specification of both exact and non-exact real-time synchronous/GALS programs.

3) The work sheds light on a key insight relating real-time and logical-time – the resolution of any real-time statement is dependent on the worst case reaction (logical tick) time – an aspect that seems to be absent in the current literature.

All aforementioned contributions require a single assumption that real-time analysable execution platforms are available that allow computation of the timing bounds of SC-SystemJ programs. This assumption is necessary for all hard real-time system implementations [83].

The rest of the chapter is organised as follows: Section 6.2 gives the background on a technique for mapping logical time to physical time, which is required for the rest of the chapter. Section 6.3 introduces real-time mechanisms in the GALS paradigm and describes their compilation into logical time. Section 6.4 sketches the correctness proofs for the introduced algorithms. Section 6.5 gives the experimental results for a set of real-time applications. Section 6.6 discusses the advantages and limitation of the proposed approach. Section 6.7 positions this work in relation to other techniques that introduce real-time in similar languages.

### 6.2 Worst case reaction time and best case reaction time for mapping logical time to physical time

The concept of perfect synchrony hypothesis and worst case reaction time (WCRT) has already been explained in previous chapters (see Section 5.2.1). In this chapter, the concept of best case reaction time (BCRT) will also be used to introduce real-time waits in the SC-SystemJ language. Formally, let $\{\Delta_1, \Delta_2, \ldots, \Delta_N\}$ be the set of all possible
6.2 Worst case reaction time and best case reaction time for mapping logical time to physical time

\{
    input signal A;
    int signal counter op+ = 0;
    while(true){
        abort(immediate A){
            while(true) pause;
        } // abort end
        emit 0;
        if(#counter != 0)
        { /* data computation leading to WCRT */}
        #counter = #counter + 1;
    } // loop end
} // SC-SystemJ CD end

(a) Simple SC-SystemJ program

\begin{figure}[h]
\centering
\begin{tabular}{c|c|c|c|c|c|c}
  & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
logical time & & & & & & \\
\hline
A & A & & & & & \\
\hline
O & O & & & & & \\
\hline
physical time & 1_s & 2_s & 3_s & 4_s & 5_s & 6_s \\
\hline
1_e & 2_e & 3_e & 4_e & 5_e & 6_e \\
\hline
A & O & A & \rightarrow W & A & O \\
\end{tabular}
\caption{Simple SC-SystemJ example and corresponding MoC}
\end{figure}

(b) Mapping to physical time

Figure 6.4: Simple SC-SystemJ example and corresponding MoC

reaction times for some CD. Then \( WCRT = \Delta_i \) where \( \Delta_i \geq \Delta_x, \forall x \in \{1, 2, \ldots, N\} \), similarly, \( BCRT = \Delta_j \) where \( \Delta_j \leq \Delta_x, \forall x \in \{1, 2, \ldots, N\} \), \( i \) and \( j \) may be the same.

For illustrating both WCRT and BCRT, consider a SC-SystemJ clock-domain shown in Figure 6.4a. This clock-domain is waiting for an input signal and is continuously producing logical ticks as shown in Figure 6.4b. Once A is received at tick 1 or 4, output O is emitted to the environment instantaneously. WCRT and BCRT are denoted as \( \mathbb{W} \) and \( \mathbb{B} \), respectively. Figure 6.4b also shows a continuously running physical clock (analogous to a clock in digital hardware); the numerical annotations on the rising edge of the clock mark the logical ticks. The subscripts \( s \) and \( e \) for the numbers show the beginning and end of the logical ticks (BOT and EOT), respectively. The end of a logical tick and the beginning of the next logical tick happen together. Figure 6.4b shows the mapping of the logical time to the physical time. For example, the logical tick 1 starts at the first rising edge and completes at the second rising edge of the physical clock, whereas logical tick 4 starts at the 4th rising edge, but finishes at the 6th rising edge of the physical clock. This difference, in duration, can be accounted for by the difference in data computations.
signal T;
abort(T){
    int signal x op+ = 0;
    while(true){
        #x = #x + 1;
        if(#x == d) emit T; //wait for "d" ticks
        pause;
    } // d is the number of logical ticks calculated
} // d is the number of logical ticks calculated

Figure 6.5: The rewrite of wait_inbetween construct

required when signal A is input from the environment (see Figure 6.4a).

6.3 Introducing Real-Time in SC-SystemJ

This section introduces a derived statement called wait_inbetween (M..N), built from kernel constructs in the SC-SystemJ language, for real-time control. The other two constructs wait_atleast(M) and wait_exact are variants of this derived construct.

6.3.1 Semantics of the wait_inbetween statement

Given a SC-SystemJ program: wait_inbetween(M..N); p, where $M \in \mathbb{Q}^>0$ and $N \in \mathbb{Q}^>0$, statement $p$ is executed after real-time postponement of $\tau$ units, such that, $M \leq \tau \leq N$.

Two variants of the derived statement wait_inbetween are also introduced here:

1) Given a SC-SystemJ program wait_atleast(M); p, where $M \in \mathbb{Q}^>0$, statement $p$ is executed after real-time postponement of $\tau$ units, such that $M \leq \tau < \infty$. It is important to note that the lower bound of the wait construct $M$ is not an exact postponement, but rather the control is allowed to proceed to the next statement any time after waiting for at least $M$ time units.

2) Given a SC-SystemJ program wait_exact(M); p, where $M \in \mathbb{Q}^>0$, statement $p$ is executed after real-time postponement of $\tau$ units, such that $\tau = M$. In this variant, the waiting time is exact.
6.3 Introducing Real-Time in SC-SystemJ

Algorithm 6.1: Finding the value of $d$

<table>
<thead>
<tr>
<th>Data: WCRT, BCRT, $M \in \mathbb{Q}^{&gt;0}$, $N \in \mathbb{Q}^{&gt;0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: $d$</td>
</tr>
<tr>
<td>Let $l_1 \leftarrow \lceil \frac{M}{\text{WCRT}} \rceil$;</td>
</tr>
<tr>
<td>Let $l_2 \leftarrow \lceil \frac{M}{\text{BCRT}} \rceil$;</td>
</tr>
<tr>
<td>Let $u_1 \leftarrow \lfloor \frac{N}{\text{WCRT}} \rfloor$;</td>
</tr>
<tr>
<td>Let $u_2 \leftarrow \lfloor \frac{N}{\text{BCRT}} \rfloor$;</td>
</tr>
<tr>
<td>Let $F : (l_1, u_1) \rightarrow S_1$;</td>
</tr>
<tr>
<td>Let $F : (l_2, u_2) \rightarrow S_2$;</td>
</tr>
<tr>
<td>Let $D \leftarrow S_1 \cap S_2$;</td>
</tr>
<tr>
<td>return (some $d \in D$);</td>
</tr>
</tbody>
</table>

6.3.2 Rewriting the wait_inbetween statement

The introduced wait_inbetween construct is not a kernel statement, but a derived construct built from the kernel constructs in SC-SystemJ. Figure 6.5 gives the rewrite of the wait_inbetween construct to kernel constructs.

The important observation is that real-time is converted into logical time via the pause construct. The rewrite basically maps real-time back to the elegant logical notion of time; the program waits a certain number of logical ticks before proceeding to the next statement. The number of logical ticks $d$ to wait (Figure 6.5) is determined by the compiler statically at compile time. As will be explained in the next section, finding the value of $d$ requires a knowledge about the WCRT and BCRT of the program, and hence the execution platform.

6.3.3 Finding the number of logical ticks $d$

The procedure to find $d$ is shown in Algorithm 6.1. This algorithm is carried out for each SC-SystemJ clock-domain or a synchronous program individually. The important observation is that the reaction time for each logical tick is elastic – varying only between the BCRT and the WCRT; thus, any number of logical ticks $d$ that map to the required real-time wait_inbetween($M..N$) should be chosen in such a way that it is invariant to this elasticity.

Algorithm 6.1 takes as input WCRT, BCRT, and the lower and upper bounds of the wait_inbetween construct $M$ and $N$, respectively. First, $M$ and $N$ are divided with the BCRT and WCRT, respectively. This division gives the number of individual logical ticks, at the fastest (BCRT) and slowest (WCRT) program execution speeds, required to postpone the clock-domain (or synchronous program) by the real-time specification. To make sure that the resultant values are integers (in domain $\mathbb{N}^{>0}$), the number is always
Exact and Non-exact Real-time Waits in a GALS Language

When dividing $M$ and $N$, these functions guarantee that the resultant logical ticks result in real-time postponements between the required range $[M, N]$. Next, a function $F$ maps these calculated values to a set of consecutive integer points (values) – these points represent all the numbers of logical ticks running at the WCRT and the BCRT, respectively that satisfy the real-time wait requirements. The intersection of these two sets gives all the number of logical ticks that satisfy the real-time requirement invariant to the logical time and its elasticity.

Let us revisit the HRTCS example (ef. Figure 6.1) to illustrate the algorithm. From Figure 6.1 it is known that $M$ is 50.3 ms and $N$ is 200.3 ms, respectively. Timing analysis of the program has found WCRT and BCRT to be: 0.112 ms and 0.0334 ms, respectively. Thus, the algorithm proceeds as follows:

1) $l_1 \leftarrow \lceil \frac{50.3}{0.112} \rceil$ and $u_1 \leftarrow \lfloor \frac{200.3}{0.112} \rfloor$. $l_1 = 450$ and $u_1 = 1788$. Here, the number of logical ticks that are always running at the WCRT are calculated first and satisfy the required real-time wait period.

2) $l_2 \leftarrow \lceil \frac{50.3}{0.0334} \rceil$ and $u_2 \leftarrow \lfloor \frac{200.3}{0.0334} \rfloor$. $l_2 = 1506$ and $u_2 = 5997$. The same calculations are done for the BCRT case.

3) $S_1 = [450, 1788]$ and $S_2 = [1506, 5997]$. Then the resultant bounds are mapped to a set of consecutive integer points. Sets $S_1$ and $S_2$ represent numbers of logical ticks running at the WCRT and BCRT, respectively, that always satisfy the required real-time constraints.

4) $D = S_1 \cap S_2$, $D = [1506, 1788]$. Finally the intersection of the two sets gives the set $D$ from which any value for $d$ can be chosen.

The resultant value for $d$ gives the number of logical ticks, which can run at any physical clock-speed, bounded by the BCRT and the WCRT of the program, and still result in the desired real-time wait period. Figure 6.6 gives the solution space for the aforementioned calculation. The patterned space is the only overlapping region and hence gives the solution space for the above example. It is possible that $D$ is an empty set; this means no overlapping region could be found. In this case, there are two possible solutions: (1) relaxing the upper real-time bound in the wait statements, or (2) running the clock-domain periodically at WCRT. The next subsection elaborates these solutions in more detail.

This is an elegant solution because the technique provides real-time guarantees while preserving the essence (elastic logical tick) of synchronous and GALS MoCs of SC-SystemJ and Esterel style languages. Moreover, this technique can also be applied when there are multiple clock-domains running on a single processor sharing resources. In such case the
6.3 Introducing Real-Time in SC-SystemJ

```latex
\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{solution_space.png}
\caption{The pictorial representation of the solution for the HRTCS system}
\end{figure}
```

postponement in the number of logical ticks is calculated after scheduling clock-domains or synchronous programs has been performed.

### 6.3.4 Finding the number of logical ticks \( d \) for the variants of `wait_inbetween` statement

#### The `wait_atleast(M)` statement

Finding \( d \) for the `wait_atleast` variant can easily be accommodated. All one needs to do is find the set \( S_2 \) and choose a value from this set.

#### The `wait_exact(M)` statement

The `wait_exact(M)` variant is more interesting. Like before, one finds sets \( S_1 \) and \( S_2 \), and the intersection of the two sets gives values for \( d \). It is possible that the resultant set \( D \) is empty (also possible in the case of `wait_inbetween(M..N)`, but never possible in case of `wait_atleast(M)`). Note that \( S_1 \) and \( S_2 \) can never be empty sets and will always contain at minimum a single element. For such cases, two solutions are provided as listed below.

#### Relaxation of the upper real-time bound

The relaxation technique is shown in [Algorithm 6.2](#). This algorithm results in the smallest increase of the upper bound required for the real-time postponement to be satisfied. The algorithm takes as input the WCRT and set \( S_2 \), recall that set \( S_2 \) represents the logical ticks required to satisfy the real-time requirement at the BCRT. The first value from set \( S_2 \) is then taken and multiplied with the WCRT to get the relaxation \( N' \). The first element of set \( S_2 \) is returned as the number of logical tick \( d \). The important observation is that
Algorithm 6.2: Calculating the minimum relaxation of the upper real-time bound

\textbf{Data:} \( S_2, D, WCRT \)

\textbf{Result:} \( d \)

\textbf{if} \( D = \emptyset \) \textbf{then}

\hspace{1em} \text{let} \( j_0 \) be the first element of set \( S_2 \);

\hspace{1em} \text{let} \( N' \leftarrow WCRT \times j_0 \);

\hspace{1em} \text{let} \( d \leftarrow j_0 \);

\textbf{return} \( d \);

one needs to postpone the proceeding statement for at least \( M \) units of real-time, hence, under-approximation is out of question. The algorithm can still over-approximate, but to reduce the resultant error one should over-approximate by the least possible value, which is the lower bound of set \( S_2 \). Thus, the lower bound is considered to be the only element shared between the two sets \( S_1 \) and \( S_2 \) and accordingly, the upper real-time bound is relaxed by the multiplication of \( WCRT \) and the first element of set \( S_2 \).

For example, assume \( BCRT = 5 \) and \( WCRT = 100 \), with a required postponement of \((200..200)\). \underline{Algorithm 6.1} results in \( D \) being an empty set, since \( S_1 = [2] \) and \( S_2 = [40] \), respectively. In such a case, the resultant relaxation is: \( 40 \times 100 \) using \underline{Algorithm 6.2}, which results in a time of 4000 units. Such large over-approximations can be avoided by breaking the critical paths (worst case ticks) in the program by inserting pause constructs, just like in hardware design, which is very unsurprising considering synchronous languages have their origins in hardware.

**Periodic execution of the clock-domains**

The aforementioned technique maintains the essence of synchronous and GALS programs (the logical tick), but in doing so is unable to provide the exact real-time wait of \( M \) time units in the case of the \texttt{wait_exact(M)} statement, or is unable to provide a postponement in the region of \([M, N]\) time units in the case of the \texttt{wait_inbetween(M..N)} statement, thereby violating its semantics. The second option is to execute the clock-domain periodically at the WCRT. WCRT determines the slowest execution speed of any clock-domain with the guarantee that no input events from the environment will be missed (cf. \underline{Section 6.2}). Thus, if the clock-domain is executed periodically at the WCRT, one can choose the value of \( d \) from set \( S_1 \) since:

\[
\begin{align*}
\text{If } BCRT = WCRT \text{ then } S_1 &= S_2 \\
S_1 \cap S_2 &= S_1 \\
\therefore \text{ choose } d &\in S_1
\end{align*}
\]
This technique allows meeting the synchronous hypothesis, while at the same time guaranteeing the semantics of the \texttt{wait\_exact}(M) statement. The designer should be allowed to choose from either of these techniques (i.e. relaxation of the upper real-time bound or periodic execution of clock-domains) and hence, this option is provided via compiler switches.

### 6.3.5 The tool-chain flow

The interaction of the low-level execution platform and the described high-level technique of introducing real-time waits in the SC-SystemJ language is shown in Figure 6.7. Automatic static analysis techniques to find the WCRT/BCRT of the programs is applied after rewrites and standard compiler optimisations. To achieve this one needs timing analysable platforms, in our case the \textit{Java Optimized Processor} (JOP) [100]. It is worth pointing out that since all rewrites, including the rewrite for the wait constructs, happen before WCRT/BCRT analysis, the analysis incorporates the effect of the introduction of the wait statements. Moreover, since algorithms 6.1 and 6.2 are performed after scheduling clock-domains, and possibly reactions, the technique is also applicable for targeting platforms lacking in the support for parallel execution. The low-level analysis guarantees a precise WCRT/BCRT for the logical tick, which is then used for calculating $d$ as in Figure 6.5.

The compiler has a switch that enables over-approximation as described in Section 6.3.4 provided the logical time $d$ could not be found. If this compiler switch is not enabled, the compiler simply decides to execute the clock-domain periodically, where the period is

![Figure 6.7: The tool-chain flow of SC-SystemJ compiler with waits](image-url)
the WCRT. WCRT affects the over-approximation and the period of the clock-domain, irrespective of the solution that the system designer chooses. A large WCRT results in reduced performance and hence it is essential that the WCRT be minimized. One can optimise the WCRT of any clock-domain by breaking the critical paths using the pause construct, which will be illustrated later in Section 6.5.

6.4 Proofs of correctness, completeness and soundness

Section 6.3 described the techniques and algorithms for introducing various types of wait statements of providing real-time control in the SC-SystemJ language. This section gives a formal proof of correctness for algorithms described in Section 6.3.

Lemma 6.4.1. Algorithm 6.1 gives a value for \( d \) such that:
\[
\left\lfloor \frac{M}{BCRT} \right\rfloor \leq d \leq \left\lceil \frac{N}{WCRT} \right\rceil
\]
provided \( D \) is a non-empty set for construct \texttt{wait\_inbetween}(M..N).

Proof. This lemma is trivially true from the definition of set \( D \) in Algorithm 6.1.

Theorem 6.4.1. Given \texttt{wait\_inbetween}(M..N) Algorithm 6.1 gives a value \( d \), provided \( D \) is a non-empty set, such that for any given reaction \( \Delta \) in the range of \( BCRT \leq \Delta \leq WCRT \), \( M \leq \Delta \times d \leq N \) holds.

Proof. We use contradiction to prove the theorem.

1. Proof for case \( \Delta = WCRT \): Assume \( d \times WCRT < M \), then \( d < \frac{M}{WCRT} \). \( BCRT \leq WCRT \), by definition, hence, \( \frac{M}{WCRT} \leq \frac{M}{BCRT} \). Thus for the assumption to hold, \( d < \frac{M}{BCRT} \), which contradicts Lemma 6.4.1. Next, assume \( d \times WCRT > N \), then \( d > \frac{N}{WCRT} \). Again, this contradicts Lemma 6.4.1.

2. Proof for case \( \Delta = BCRT \): Like above, the assumption \( d \times BCRT < M \) is trivially proven from Lemma 6.4.1. For \( d \times BCRT > N \) to hold, \( d > \frac{N}{BCRT} \) should hold. But, \( BCRT \leq WCRT \), by definition, hence, \( \frac{N}{BCRT} \geq \frac{N}{WCRT} \), which again contradicts Lemma 6.4.1.

3. Proof for case \( BCRT < \Delta < WCRT \): For \( d \times \Delta < M \) to hold, \( d < \frac{M}{\Delta} \) should hold. By definition, \( \Delta > BCRT \), thus, \( \frac{M}{\Delta} < \frac{M}{BCRT} \), hence, \( d < \frac{M}{BCRT} \) should hold, which contradicts Lemma 6.4.1. Next, for \( d \times \Delta > N \) to hold, \( d > \frac{N}{\Delta} \) should hold. By definition, \( \Delta < WCRT \), thus \( \frac{N}{\Delta} > \frac{N}{WCRT} \), hence, \( d > \frac{N}{WCRT} \), which again contradicts Lemma 6.4.1.
Next, we prove the completeness property.

**Theorem 6.4.2.** Given `wait_inbetween (M..N)` Algorithm 6.1 gives a value \(d\), provided \(D\) is a non empty set, such that for all given reaction times \(\Delta\) in the range of \(BCRT \leq \Delta \leq WCRT\), \(M \leq \Delta \times d \leq N\) holds.

**Proof.** Follows from **Theorem 6.4.1**.

Next, we prove the soundness property of Algorithm 6.1.

**Theorem 6.4.3.** Given `wait_inbetween (M..N)` Algorithm 6.1 does not give a value \(d\), such that for all given reaction times \(\Delta\) in the range of \(BCRT > \Delta > WCRT\), \(M \leq \Delta \times d \leq N\) holds.

**Proof.** Soundness is trivially proven knowing that there exists no \(\Delta < BCRT\) or \(\Delta > WCRT\) by definition (cf. Section 6.2).

**Lemma 6.4.2.** Algorithm 6.2 gives a value for \(d\) such that: \(N' = WCRT \times d\) and \(d = \lceil \frac{M}{BCRT} \rceil\).

**Proof.** This lemma is trivially true from the definitions of \(N\) and \(d\) in **Algorithm 6.2**.

**Theorem 6.4.4.** Given `wait_inbetween(M..N)` Algorithm 6.2 gives a minimum relaxation of the upper real-time bound \(d\), provided \(D\) is an empty set, such that for any given reaction time \(\Delta\) in the range of \(BCRT \leq \Delta \leq WCRT\), \(M \leq \Delta \times d \leq N'\) holds.

**Proof.** We again proceed by contradiction.

1. Proof for case \(\Delta \leq WCRT\): Assume \(\Delta \times d > N'\), then from **Lemma 6.4.2** \(\Delta > WCRT\), which contradicts the definition of \(WCRT\).

2. Proof for case \(\Delta = BCRT\): Assume \(BCRT \times d < M\), then \(d < \frac{M}{BCRT}\). \(d = \lceil \frac{M}{BCRT} \rceil\) from **Lemma 6.4.2** thus contradicts \(d < \frac{M}{BCRT}\).

3. Proof for case \(\Delta > BCRT\): Assume \(\Delta \times d < M\), then \(d < \frac{M}{\Delta}\). By definition, \(\Delta > BCRT\), thus, \(\frac{M}{\Delta} < \frac{M}{BCRT}\). Again \(d = \lceil \frac{M}{BCRT} \rceil\), then \(\frac{M}{\Delta} < d\), hence contradiction.

Correctness, completeness, and soundness for `wait_atleast (M)` and `wait_exact (M)` are dependent on **Theorem 6.4.1** because these are variants of the `wait_inbetween (M..N)` statement.
Table 6.1: An overview of the benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Delay constructs used in each CD</th>
<th># of CDs</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HRTCS</td>
<td><code>wait_inbetween(50.3..200.3 ms)</code> <code>(CD1)</code></td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>Motor</td>
<td><code>wait_exact(2.4 ms)</code> <code>wait_exact(1.667 ms)</code> <code>wait_exact(0.05 ms)</code> <code>wait_exact(0.3 ms)</code> <code>wait_exact(0.733 ms)</code></td>
<td>1</td>
<td>116</td>
</tr>
<tr>
<td>Printer</td>
<td><code>wait_inbetween(2.7..10.6 ms)</code> <code>(CD1)</code> <code>wait_inbetween(1.2..30.3 ms)</code> <code>(CD2)</code> <code>wait_inbetween(5.6..100.2 ms)</code> <code>(CD2)</code></td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>AECS</td>
<td><code>wait_atleast(10000 ms)</code> <code>(CD1)</code> <code>wait_exact(10000 ms)</code> <code>(CD2)</code> <code>wait_exact(10000 ms)</code> <code>(CD2)</code></td>
<td>2</td>
<td>267</td>
</tr>
</tbody>
</table>

6.5 Experimental results

A number of experiments have been carried out on a set of applications with real-time constraints. The characteristics of the benchmark set including required real-time waits and program sizes are shown in Table 6.1.

HRTCS is the human response system described in Section 6.3.1. HRTCS consists of 2 clock-domains, but only one with real-time wait statement.

Motor is a stepper motor controller from [101], used for producing monochrome images on paper by correct application of current to a print head (a row of resistors). In this example, duration of the current applied is controlled by the feedback logic using real-time waits, which are expressed using reactive constructs. It is a synchronous program with 5 exact wait statements.

Printer is an example from [102], which models a printer and a spooler in timed CSP. Its operation can be described as follows: as soon as the spooler receives a print job it passes it to the printer through a channel where the actual printing process takes place. Each spooler and printer process is mapped to a separate clock-domain. The spooler clock-domain sends a job to the Printer clock-domain between 2.7 and 10.6 ms after receiving it from the environment (e.g. a user). On the other hand, the Printer clock-domain notifies the environment that it starts printing between 1.2 and 30.3 ms once it receives the job from the Spooler clock-domain. The actual print job is started between 5.6 and 100.2 ms after the notification. In this example, bounded wait ensures that both channel communication and print jobs should occur within the specified amount of time.

AECS (Access and Environmental Control System) is an application that controls an intelligent room environment as described in [30]. AECS has 2 clock-domains; with one and two wait statements, respectively. The system is designed to measure how long the entrance door to the room is open and to detect unknown entries (i.e. intruders). For instance, the system triggers an alarm 10 seconds after detection of an intruder or
generates a beep sound when an entrance door is kept open for the same amount of time. In the original AECS the system needed to interact with two external timers through input/output signals in order to wait for a specific amount of time (e.g. TimerTriggered in Figure 6.8a). When the timer expires it notifies the system through the input signal TimeOut, which is polled using the await statement. The program has been modified such that the wait constructs replace external timers and corresponding interface (I/O) signals, which is shown in Figure 6.8b. With the new wait constructs these timers are not necessary as the real-time waits are expressed directly in the SC-SystemJ code.

All experiments are carried out on the Java Optimized Processor (JOP) running at 60 Mhz, which has been shown to be real-time analysable. WCRT and BCRT of these benchmark applications are statically computed using the real-time analysis tools provided by the JOP design framework. Table 6.2 shows the result of the presented algorithm applied to each benchmark program. Note that in this experiment the compiler was allowed to relax the upper real-time bounds for wait_exact and wait_inbetween when it failed to find a feasible solution. However, a user can prohibit such behaviour using compiler switch as described in Section 6.3.5.

As a first step, BCRT and WCRT were calculated for all examples. Given the real-time

<table>
<thead>
<tr>
<th>BCRT</th>
<th>Printer/CD1</th>
<th>Printer/CD2</th>
<th>AECS/CD1</th>
<th>AECS/CD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0334 ms</td>
<td>0.0425 ms</td>
<td>0.0190 ms</td>
<td>0.1284 ms</td>
<td></td>
</tr>
<tr>
<td>0.112 ms</td>
<td>0.1096 ms</td>
<td>0.0647 ms</td>
<td>0.3476 ms</td>
<td></td>
</tr>
<tr>
<td>156-201 ms</td>
<td>2.4-6.2 ms</td>
<td>3.7-10.6 ms</td>
<td>10000-19484 ms</td>
<td></td>
</tr>
<tr>
<td>10000-23278 ms</td>
<td>10000-23278 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.8: Replacing external timer with wait_atleast construct
wait_inbetween(M..N) provided by programmers, the required number of logical ticks \( d \) was determined by applying Algorithm 6.1. In HRTCS, the real-time delay of the range 50.3..200.3 (ms) is only satisfied when \( d \) is between 1506 and 1788 (as shown previously in Section 6.3.3). Delay 2.7..10.6 ms is satisfied in Printer/CD1 with \( d \) of 142..163. Similarly, Printer/CD2 satisfies real-time ranges of 1.2..30.3 ms and 5.6..100.2 ms when \( d \) is 46..617 and 213..2041, respectively. It is then the compiler’s job to statistically choose one of the values in this range.

On the other hand, Algorithm 6.2 is applied to those examples where relaxation was required to determine the minimum upper real-time bound \( N \) such that there exists a solution for \( D \) in Algorithm 6.1. This algorithm gives a unique number for \( d \) such that \( d = \text{Min}(S_2) = \text{Max}(S_1) \). The compiler, therefore, will chose \( d \) based on the newly computed \( N \) for the examples Motor and AECS.

### 6.5.1 Improving results

The previous results show that for HRTCS and Printer the compiler was able to find \( d \) without relaxing the upper bound of wait statements. Figure 6.9 shows the accuracy of the results by comparing between the programmer specified upper bounds (Table 6.1) and the actual bounds obtained using the introduced techniques (Table 6.2). Here, the difference factor for each benchmark is obtained simply by division of the relaxed upper real-time bound by the original upper real-time bound. Therefore, a smaller difference factor means less deviation from the original upper real-time bound.

There are two influential factors in determining the relaxation: (1) execution speed of target platforms as clock frequency of the processor directly impacts the value of WCRT, and (2) the amount of data computation that needs to be performed for each clock-domain tick. Programmers can write a program such that large data (or control) computations in a single logical tick increases WCRT of the program. Assuming that the speed of the execution platform is fixed, the difference factor can only be reduced by modifying the
6.5 Experimental results

benchmarks programs such that the critical paths are broken up using additional \texttt{pause} statements, thereby reducing the WCRT. Figure 6.10 shows segments of code from AECS whose WCRT is reduced by such a technique.

In Figure 6.10a the program simply checks for the id of a person entering or exiting an intelligent room through the signals \texttt{EntryRequest} and \texttt{ExitRequest} (lines 3 and 5), and verifies the id using the data computation written in the \texttt{extern} function (\texttt{isValidRequest} at line 6). It was identified that this was the critical execution path accounting for the WCRT. In this case, a \texttt{pause} statement was inserted (line 5 in Figure 6.10b) between the \texttt{present} and the \texttt{if} statements (lines 1 and 7) such that these statements are executed in two different ticks. The same technique is applied to the Motor example, which in this case is purely control-dominated, to reduce the WCRT of the program. Another option to shorten the WCRT is to reduce execution time of the data computations used in each \texttt{extern} function. However, this is out of the scope of this chapter, hence not discussed here.

The result of the upper bound relaxation with reduced WCRT is shown in Figure 6.11. The upper bounds for wait statements in AECS/CD2 decreased by 11.81%, while for
the Motor example the reductions ranged from 6.42% to 8.05%. This is as expected since the amount of computation required in control-flow is less demanding than in data computation, hence breaking the critical paths in Motor did not reduce WCRT as much as AECS/CD2. Therefore, one can see from the result that the size of the relaxation depends on how designers write their program as well as the type of the application (i.e. data or control dominated).

6.6 Discussion

This section is dedicated to discussing the details of the real-time wait construct semantics, its advantages and limitations in a reactive setting.

6.6.1 Programming a periodic reaction using the wait_exact construct

```c
while(true) {
    while(true) {
        abort(T) {
            int signal x op+ = 0;
            while(true) {
                #x = #x + 1;
                if (#x == d) emit T;
                pause;
            }
            // extra pause
            pause;
            emit S;
            emit S;
        }
        while(true) {
            wait_exact(1 ms);
            //extra pause
            pause;
            emit S;
        }
    }
}
```

(a) Original program (b) Rewritten program

Figure 6.12: Extra pause statements in periodic tasks

One should be aware of the interaction of wait_exact and other pause statements in the body of the reaction (or clock-domain). For example, consider the periodic reaction code snippet in Figure 6.12a and its rewrite in Figure 6.12b.

The system designer needs to be aware that just introducing a wait_exact construct
does not make the reaction (or clock-domain) periodic. The code snippet in Figure 6.12a does not emit signal S every 1 ms, because of the extra pause statement. Introducing additional pauses inside the while loop introduces more logical ticks to be consumed by the reaction before it can re-enter the wait_exact statement, consequently increasing the real-time period of this reaction.

### 6.6.2 Need for time analysable platforms

The solution for introducing real-time as first class language constructs in this work rely on using execution platforms that allow computation of the WCRT and the BCRT of the program.

**Interaction of timers and reactive constructs**

Preemption plays an important role in reactive languages. One needs to carefully consider the interplay of wait construct semantics with the preemption semantics of reactive languages. Previous attempts at incorporating postponement using external timers has only been partially successful because of the complex interplay between real-time and preemption. Consider the simple example in Figure 6.13 which models real-time using external timers as in [103].

```plaintext
suspend(S) {
    emit START_TIMER(10);
    await TIMER;
    emit 01;
}
```

Figure 6.13: Interaction of preemption and external timers

As identified in [101] suspend does not play well with the external timer. The program in Figure 6.13 sends a signal to an external timer and waits on the timeout for 10 ms to pass by. Consider what happens when signals S and TIMER occur in the same logical tick, the await statement is never executed (due to suspend) and hence, this results in a deadlock. Such problems are completely avoided in the technique presented in this work because the real-time wait constructs are converted into a sequence of logical ticks using the pause statement which does not rely on any external event(s) coming from the environment (e.g. timer). Furthermore, synchronous languages have always been targeted at real-time analysable platforms and constrained environments [104, 17]. One can use our technique within such a setting with ease.
{ 
  //reaction R1
  emit START_TIMER(1 ms);
  await (TIMER);
  emit O1;

  //synchronous parallel

  //reaction R2
  // do something with heavy computation
}

Figure 6.14: Resolution of external timers

The timer resolution problem

There is yet another problem with timer based systems. First, consider the code snippet in Figure 6.14. On a cursory look this example should work fine. An external timer is started in reaction R1, which counts down from 1 ms. Once this time has elapsed, a signal TIMER is generated from this external timer, which should emit O1 in turn. There is no suspend construct encapsulating the timer and hence all seems fine. Next consider what happens to the whole clock-domain, including reaction R2; while reaction R1 is waiting for the TIMER signal, reaction R2 is making progress. Input signals can only be captured from the environment at logical tick boundaries (cf. Section 6.2). In Figure 6.14, assuming reaction R2 performs some large data computation, the length of the tick is determined by R2. Consider what happens if the external timer generates the TIMER signal, but the system is still performing computation and the logical tick has not yet finished – the TIMER signal will be missed. Thus, even in the absence of the suspend construct, it is not guaranteed that the program will capture the external signal. In fact, the WCRT needs to be smaller than 1 ms for the above code snippet to preform as expected.

In general it is observed that the WCRT determines the lowest resolution of any real-time construct whether it be external timer independent, like the proposed technique, or external timer dependent, like above. This analysis is one of the key insights for proposing the solutions described in this chapter.

Interaction of channel communication and wait constructs

Channels, used for communication between reactions in asynchronously running clock-domains, are an addition in the SC-SystemJ language. Like interaction of preemption and waits, conversion of real-time wait constructs to logical ticks also interacts well with channel rendezvous because the semantics of interaction are well defined (Section 3.5.2). More importantly, one needs to consider the interplay of channel communication and
6.7 Other approaches to implementing real-time constructs in the reactive languages

WCRT and BCRT analysis. Since channel communication does not stop logical time, WCRT and BCRT are unaffected by channel communication.

6.7 Other approaches to implementing real-time constructs in the reactive languages

A number of studies have been performed in the attempt to integrate real-time directly as a programming construct in GALS as well as synchronous reactive languages. The most prominent works in this are by Shyamsundar [103] and Bourke et al. [101]. Shyamsundar incorporates real-time using external timers in Communicating Reactive Processes (CRP), which, like SC-SystemJ, is an extension of Esterel with asynchrony. But, as mentioned in Section 6.6.2, external timers do not interact well with preemption constructs in these languages. Bourke et al. [101] introduce real-time as first class constructs in the Esterel language. They, like the work presented here, introduce real-time wait statements called delay as first class Esterel programming constructs, and translate them into Esterel kernel statements. However, they do not translate wait statements into pause statements directly. Instead, logical ticks are generated by using the abstract notion of event and sample, (i.e. execution period) platform dependent timers. This notion makes it a complex and inflexible solution since the number of platform timers with certain resolutions need to be determined and present on the system for the solution to be realisable. The solution presented here does not require external timers (like [103]) or the logical notion of timers (like [101]). Instead, the real waiting time in number of logical ticks is calculated after allocation and scheduling of clock-domains on a target execution platform; it gives compiler developers the chance to optimise for many different criteria such as computation time without worrying about or violating the real-time requirement specified using the wait statements.

There are also other works exist. Logothethis et al. [105] incorporate real-time statements in Esterel using pause statements in an Esterel variant called Quartz. Nevertheless, it is targeted at studying timing properties using model checkers rather than implementation. The solution introduced by Bertin et al. [106] translates real-time pragmas (annotations) into timed automata [107] for model-checking real-time deadlines. Our approach on the other hand removes all continuous time elements and produces a single discrete time model thereby making the solution amenable to existing functional and real-time verification tools developed for reactive languages, and is more in the spirit of logical time. Finally a major difference between Bertin et al. [106] and the technique presented in this chapter is that their approach requires programmers to annotate the time taken
by ‘C’ code fragments manually. This is extremely error prone; we use static low-level program analysis to find out such times automatically.

To conclude, this chapter describes a novel way to introduce real-time in GALS languages, in particular SC-SystemJ, and their subset, synchronous languages like Esterel. The fundamental idea is to convert the real-time wait constructs into logical waits (pause construct) that interact well with the other constructs in these languages, especially pre-emption and channel communication. The introduced technique does not need or use external timers to introduce these real-time wait constructs, instead it depends upon an underlying timing analysable platform, thereby resolving the problems of interaction between external timers and preemption. Moreover, the introduced technique does not require specific timer resolutions, thereby making the solution elegant (in the spirit of GALS/synchronous programming) and flexible.
Conclusions and Future Work

Many of today’s embedded systems are safety-critical systems where failure can cause financial or life-threatening damage. Therefore, the design of such embedded systems imposes the strict requirement to meet both functional and real-time properties. Moreover, with increasing complexity of these embedded systems, which are becoming more parallel and distributed in nature, the main attraction has been towards employing formal languages and analysis techniques for verifying the correctness of the system before its deployment. Unlike traditional programming languages such as C/C++ and Java, the programming features in formal languages, such as concurrency, reactivity, and hierarchical composition etc., are based on rigorous mathematical foundations. This enables a correct-by-construction design paradigm as the same program can be used for both verification and code generation without the need to reinterpret the originally designed model for implementation.

This thesis introduces a methodology to develop and verify safety-critical programs using SC-SystemJ, which is a subset of the reactive language SystemJ, based on the formal *Globally Asynchronous Locally Synchronous* (GALS) Model of Computation (MoC). On the top-level of the SC-SystemJ program there can be one or more *clock-domains* that run asynchronously with each other. Each clock-domain follows the synchronous paradigm, which describes its behaviour as a discrete sequence of events called *ticks*. A clock-domain can further be hierarchically composed of synchronous and concurrent processes called *reactions* whose control and data oriented executions occur instantaneously.
in lock-step with the clock-domain tick. Objects called *signals* and *channels* are used for communication amongst the reactions within and between clock-domains, respectively. Signals, once emitted, have their statuses set to true and are broadcasted to all other reactions that belong to the same clock-domain. The signal statuses are made visible for only a single tick. Channels, on the other hand, are used to perform CSP-style message passing by synchronising reactions that run within two asynchronously running clock-domains.

Although the original SystemJ language was developed based on formal semantics, it was difficult for programmers to formally verify the functional correctness of their programs because the verification methodology for SystemJ programs had not yet been adequately investigated. The work presented in this thesis has focused on introducing methods to verify programs developed in the newly introduced SC-SystemJ language for both functional and real-time correctness. In addition, the language has also been extended to make it more suitable for developing real-time computing systems while still preserving the discrete time model of the GALS and synchronous MoCs.

### 7.1 Main contributions

The detailed contributions of this thesis are:

1. **Semantics of the SC-SystemJ language based on big-step transition for developing safety-critical and real-time computing systems**: [Chapter 3](#) presented a subset of the original SystemJ language called SC-SystemJ used in the new tool-chain flow. In this approach, each SC-SystemJ clock-domain is compiled into a Linear Temporal Logic (LTL) formula, which is a transition system of the clock-domain. This LTL formula is then compiled into a Labelled Generalized Büchi Automata (LGBA) that accepts infinite inputs. A novel algorithm is applied to generate a deterministic Mealy automaton from the LGBA. This process is done for all clock-domains in the SC-SystemJ program. The result is a network of Mealy automata, which is used for generating back-end code such as C/Java. Unlike the original Structural Operational Semantics (SOS) based approach for compilation, which is used for defining the behaviour of the SystemJ program based on fine grained *micro-steps*, the new approach adopts *big-step* semantics that translates the program into a network of finite state machines. This approach is more amenable to existing verification techniques, such as model checking, which are based on automata theories. Lastly, a new signal communication model is introduced for synchronous reactions which delays any reaction to internally emitted signals by one tick. This simplifies Worst Case Reaction Time (WCRT) analysis for individual clock-domains.
2. A new correct-by-construction tool-chain flow for ensuring functional correctness of the SC-SystemJ program from specification to implementation: Chapter 4 introduced a new tool-chain flow for verification of the SC-SystemJ program before generation of executable code. The correctness of the program is verified using a model checker whose input is produced directly from the network of Mealy automata representing the SC-SystemJ program (Chapter 3). The program can optionally be closed by introducing a model of a plant (i.e. the environment) and composing it with the SC-SystemJ clock-domain using the synchronous parallel operator ($||$). This allows designers to verify functional correctness of the program with certain assumptions on input patterns if the behaviour of the environment is known at design time. Inputs can still be left open if needed. In this open-loop verification approach the model checker will verify the properties on all possible execution traces of the SC-SystemJ program. The model checker verifies functional correctness of the SC-SystemJ program against a set of LTL formulas, which can be either liveness or safety properties, given by the designer.

Sending and receiving operations on channels are implemented using a new four phase handshake mechanism using signals. This allows verification of the SC-SystemJ program without breaking the semantics of reactivity by blocking clock-domains at the logical tick level rather than halting. Once all properties (e.g. liveness and safety) are satisfied, executable code is generated. The tool-chain produces executable code from the exact same model on which the properties are verified, hence fulfilling the What You Prove is What You Execute (WYPIWYE) paradigm. Lastly, the benchmark results show that the new automata-based compiler generates on average 3.3x smaller and 2.1x faster code than the original AGRC-based compiler.

3. Introducing a response time analysis technique for SC-SystemJ and GALS programs: Chapter 5 presented a technique to guarantee real-time constraints between input and output events of the SC-SystemJ program. In this approach each SC-SystemJ clock-domain is converted into a graph whose edge is annotated with transition times obtained from the traditional WCRT analysis. This graph as well as the program specified real-time constraints, are then formulated as a Satisfiability Modulo Theories (SMT) problem, which can be solved by a SMT solver. The proposed approach is applicable to the scheduling of GALS programs for different target architectures with single or multiple processor systems by guaranteeing execution of clock-domains with exclusive access to the underlying processor. The output of the solver is a schedule with release times for each program transition that guarantees the response time constraints. The complexity of the proposed approach increases
as the number of processor cores decreases, thereby making it scalable. It has been shown in the benchmark tests that the response time could be reduced up to 27% by exploiting the inherent parallelism with increasing number of processors.

4. Enhancing the expressiveness of formal GALS and synchronous language by adding real-time constructs: Chapter 6 introduced new SC-SystemJ constructs, which provide exact and non-exact real-time control mechanisms. In particular, the program can be postponed from proceeding to the following statement for a minimum of $M$ and a maximum of $N$ time units by using the wait_inbetween($M..N$) statement. The wait_inbetween statement is rewritten into a SC-SystemJ temporal loop, which consumes logical ticks. The compiler calculates the number of logical ticks $d$ for a clock-domain to wait based on its Best Case Reaction Time (BCRT) and Worst Case Reaction Time (WCRT). The resultant number of ticks $d$ that satisfies real-time wait between $M$ and $N$ time units such that it is invariant to the elasticity of the reaction time of the clock-domain. However, it is possible that the compiler cannot find the solution $d$. In this case, the upper real-time bound $N$ can be relaxed to the smallest required real-time postponement. The technique has been extended to implement two other real-time wait constructs: wait_atleast($M$) and wait_exact($M$), which are variants of wait_inbetween($M..N$). The experimental results showed that these constructs can replace real-time delays implemented using external timers.

All the work accomplished in this thesis forms the basis for using the SC-SystemJ language in real-time and safety-critical applications. Although the major problems have been examined and addressed in this thesis, there are still a number of possible improvements that can be made. These are discussed in the following section.

7.2 Possible future research directions

This section highlights future research opportunities as a continuation to the work presented in this thesis.

1. Optimising the compiler: It was shown that the new SC-SystemJ compiler did not face memory constraint problems when compiling SC-SystemJ programs. However, the runtime complexity can grow exponentially due to the combinatorial explosion of states in synchronous parallel reactions during the generation of the LGBA. Yet, it is often not necessary to consider all the possible state compositions in many cases because synchronous reactions execute in lock-step. Therefore, the runtime
of the compilation can be reduced by optimising the generated LTL formulas from
the SC-SystemJ clock-domains such that the unreachable state compositions are
pruned before passing them to the tableau algorithm. Finally, the implementation
of the tableau algorithm should be revisited and improved in order to reduce the
time spent on the generation of the LGBA.

2. **Model-based automatic plant generation for verification of SC-SystemJ programs:**
The current approach to closed-loop verification of SC-SystemJ programs assumes
that the designers provide plant models manually. However, this is often not the
ideal case because, in this approach, designers are responsible for verifying the
correctness of the program with respect to all possible environmental behaviours
(plants). An alternative approach would be automatically generating plant models
based on the functional properties that designers want to verify. Authors in [108]
reviewed different methods for automated generation of test-cases, which are plant
models in our case, based on the system specification. These test-cases are then
executed on the implementation (i.e. system under test) to check their conformance
to the specification. The methods reviewed in [108] are a good starting point for
carrying out this research.

3. **Implementing a real-time communication protocol for channels and signals and in-
clude it in the response time formulation:** The current tool-chain of the response
time analysis assumes that the potential delays, which can be induced by the com-
munication links among the processors, are constant and already included in the
WCRT analysis. For example, Time Division Multiple Access (TDMA) is a com-
monly used method in many real-time systems that allocates predefined time slots
to processors sharing resources. This enables calculation of the time spent by a pro-
cessor that needs to fetch a fixed amount of data from the memory, or when sending
messages over a shared bus. However, this often results in over-estimation of WCRT
because not all processors require the same access time to shared resources. Thus,
the WCRT can be minimised by allocating appropriate time slot sizes to each pro-
cessor by analysing the application’s memory or data access patterns via the shared
communication link. Existing real-time communication protocols such as Time-
Triggered Protocol (TTP) [109] can be investigated for use in our response time
analysis technique. TTP provides several features such as fault tolerance and clock
synchronisation with a small overhead in the message size.

4. **Implementing a scheduler on the target execution platform that guarantees response
time constraints:** The scheduler generated from the SMT formulation in Chapter 5
needs to be implemented on the target execution platform, be it a single proces-
sor, multi-processor shared memory or multi-processor distributed memory system.
These different processor architectures require one to implement the scheduler in different ways. In the case that all clock-domains are allocated onto a shared memory system the scheduler FSM can provide explicit signals to release the individual FSM transitions at the calculated release times under the assumption that these signals are instantaneously delivered to the SC-SystemJ program FSMs. Alternatively, a signal oscillator clock can be used with a look up table to release the individual transitions at the calculated release times. This implementation is possible due to the use of shared memory.

Implementing the scheduler on a distributed memory system is a lot more challenging. There are two possible approaches:

(a) A single FSM scheduler can be run on any one of the chosen processors. In this case, the scheduler can release the FSM transitions by sending explicit signals at the calculated release times using a single oscillator clock under the assumption that this signal is delivered instantaneously. This assumption cannot be guaranteed in all distributed-memory systems.

(b) In an alternative approach individual lookup tables corresponding to the release times for individual FSMs can be built and assigned to the processors executing these FSMs. The FSM transitions can then be released using the lookup table and the individual processor clocks. In such an implementation approach, one needs to make a single assumption that the clocks of different processors do not drift or that the drift is negligible and will not affect the schedule.

The new tool-chain flow and enhancements introduced to the SC-SystemJ language presented in this thesis enable designers to employ the GALS paradigm in the development of safety-critical and real-time software systems. The correct-by-construction (WYPIWYE) approach allows designers to verify correctness properties of the program as well as generation of executable code from the same specification, which is the SC-SystemJ program, thereby closing the gap in the design flow caused by mismatch between the model and actual implementation. In addition, the real-time wait constructs introduced in this thesis allow programmers to model various types of real-time applications that were difficult to capture with traditional synchronous and discrete-time based programming languages.
A

SC-SystemJ Grammar

⟨program⟩ := ⟨stmtlist⟩

⟨stmtlist⟩ := ⟨stmtlist⟩ ⟨stmt⟩ | ⟨stmt⟩

⟨stmt⟩ := [‘input’ | ‘output’] [⟨(datatypes)⟩ ‘signal’ ⟨symbol⟩ ‘op’ [‘+’ | ‘*’] ‘=’ ⟨number⟩] ‘;’
  | [‘input’ | ‘output’] [⟨(datatypes)⟩ ‘channel’ ⟨symbol⟩ ‘=’ ⟨number⟩] ‘;’
  | ‘C’ ⟨stmtlist⟩ ‘;’
  | ⟨stmt⟩ ‘[’ ⟨stmt⟩ ‘]’
  | ‘if’ ‘C’ ⟨expr⟩ ‘)’ ⟨stmt⟩ [‘else’ ⟨stmt⟩]
  | ‘present’ ‘C’ ⟨sigexpr⟩ ‘)’ ⟨stmt⟩ [‘else’ ⟨stmt⟩]
  | ‘abort’ ‘C’ [‘immediate’] ⟨sigexpr⟩ ‘)’ ⟨stmt⟩
  | [⟨symbol⟩ ‘:’] ‘await’ ‘C’ [‘immediate’] ⟨sigexpr⟩ ‘)’ ⟨stmt⟩
  | ‘suspend’ ‘C’ [‘immediate’] ⟨sigexpr⟩ ‘)’ ⟨stmt⟩
  | ‘emit’ ⟨symbol⟩ ‘;’
  | [⟨symbol⟩ ‘:’] ‘send’ ⟨symbol⟩ [‘(’ ⟨dataexpr⟩ ‘)’] ‘;’
  | [⟨symbol⟩ ‘:’] ‘receive’ ⟨symbol⟩ ‘;’
  | ‘while’ ‘C’ [‘true’] ‘)’ ⟨stmt⟩
  | ‘pause’ ‘;’
  | ⟨datastmts⟩

⟨datatypes⟩ := [‘int’ | ‘short’ | ‘byte’]

⟨number⟩ := ⟨digit⟩+
\[ (\text{digit}) \quad ::= \quad '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \mid '0' \]

\[ (\text{alphabet}) \quad ::= \quad 'a' \mid 'b' \mid 'c' \mid 'd' \mid 'e' \mid 'f' \mid 'g' \mid 'h' \mid 'i' \mid 'j' \mid 'k' \mid 'l' \mid 'm' \mid 'n' \mid 'o' \mid 'p' \mid 'q' \mid 'r' \mid 's' \mid 't' \mid 'u' \mid 'v' \mid 'w' \mid 'x' \mid 'y' \mid 'z' \mid 'A' \mid 'B' \mid 'C' \mid 'D' \mid 'E' \mid 'F' \mid 'G' \mid 'H' \mid 'I' \mid 'J' \mid 'K' \mid 'L' \mid 'M' \mid 'N' \mid 'O' \mid 'P' \mid 'Q' \mid 'R' \mid 'S' \mid 'T' \mid 'U' \mid 'V' \mid 'W' \mid 'X' \mid 'Y' \mid 'Z' \]

\[ (\text{symbol}) \quad ::= \quad (\text{alphabet}) \mid (\text{alphabet}) \mid (\text{digit})^* \]

\[ (\text{expr}) \quad ::= \quad (\text{reldataexpr}) \mid '!' \quad (\text{expr}) \mid (\text{expr}) \mid '1' \quad (\text{expr}) \mid (\text{expr}) \quad \\& \quad (\text{expr}) \mid '!' \quad (\text{expr}) \]

\[ (\text{sigexpr}) \quad ::= \quad (\text{symbol}) \mid '!' \quad (\text{sigexpr}) \mid (\text{sigexpr}) \mid '1' \quad (\text{sigexpr}) \mid (\text{sigexpr}) \quad \\& \quad (\text{sigexpr}) \mid '!' \quad (\text{sigexpr}) \]

\[ (\text{datastmts}) \quad ::= \quad '(' \mid ';' \mid '#' \quad (\text{symbol}) \quad '=' \quad (\text{dataexpr}) \quad ';' \mid \text{extfunc} \quad ';' \]

\[ (\text{reldataexpr}) \quad ::= \quad (\text{dataexpr}) \quad (\langle \text{\textless} \mid '!' \mid '=' \mid '<=' \mid '>' \mid '>=' \mid '==') \quad (\text{dataexpr}) \]

\[ (\text{dataexpr}) \quad ::= \quad (\text{dataexpr}) \quad (\langle '+' \mid '-' \mid '@' \mid '*' \mid '/' \mid '<' \mid '<=' \mid '>' \mid '<>' \mid '!=' \mid '==' \rangle \quad (\text{dataexpr}) \mid '!' \quad (\text{symbol}) \mid 'C' \quad (\text{datatypes}) \quad ')' \mid '!' \quad (\text{dataexpr}) \quad (\text{number}) \mid \text{extfunc} \]

\[ (\text{extfunc}) \quad ::= \quad '\text{extern}' \quad (\text{symbol}) \quad 'C' \quad [ \quad (\text{dataexpr}) \quad ',', \quad (\text{dataexpr})^* \quad ] \quad ')' \]
List of Publications

The techniques presented in this thesis are based on the following research articles:

**B.1 Published Papers**


References


