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.
Contributions Towards Dynamic Intelligent Software Systems

Udayanto Dwi Atmojo

Supervisors: Prof. Zoran Salcic
Dr. Kevin I-Kai Wang

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering, the University of Auckland, 2017
Abstract

The rapid advance in computing technology opens new opportunities in allowing the development of more complex distributed systems. In case of distributed systems with particular requirements like industrial manufacturing systems, concurrent software behaviours govern industrial machines (to give a certain level of intelligence) and need to dynamically change and reconfigure during runtime to satisfy manufacturing or operational requirements. These distributed systems that are governed by intelligent software behaviours and may dynamically change during runtime can be referred to as dynamic intelligent software systems (DISS). The programming of these systems may become more difficult and challenging for programmers as they grow in scale.

To identify the design and programming requirements of DISS, a motivating example of DISS is presented. The identified design and programming requirements of DISS are used to assess the capability of the state of the art programming approaches for DISS in satisfying these requirements. Based on the assessment, none of the state of the art is capable of completely satisfying the design and programming requirements of DISS.

This thesis proposes a novel programming paradigm called Service Oriented SystemJ (SOSJ) which combines both the service oriented architecture (SOA) paradigm and a formal model of computation (MoC)-based programming language SystemJ to better satisfy the design and programming requirements of DISS. The integration introduces new features, which consist of Beacon, Advertisement, Request for Advertisement, Discovery, Discovery Reply, and Notify, to achieve the SOA protocol in SOSJ. The integration of SOA paradigm introduces the concepts of service provider (software behaviour/entity which provides services) and service consumer (software behaviour/entity which uses services) into SystemJ. SOSJ uses centralized service registry to store the service description of services provided by service providers, where service consumer can perform service discovery on to obtain the list of services.

The capability of SOSJ in satisfying the design and programming requirements is further strengthened with functionalities for programmers to easily handle dynamic behaviour (i.e. dynamic creation, suspension, resumption, termination, and migration) of clock domains (mutually asynchronous software behaviours individually composed of reactions). In line of the dynamic creation, suspension, resumption, termination, and migration handling functionalities,
SOSJ clock domain (CD) macro-states are introduced, in which CD macro states may transition from one to another, forming the SOSJ CD life cycle.

Based on the proposed programming paradigm, SOSJ framework is developed by extending the original SystemJ RTS (which is programmed entirely in Java) to realize the proposed SOA protocol and dynamic behaviour handling functionalities. The capability of SOSJ is compared against two different state of the art programming approaches in satisfying the design and programming requirements of DISS. Based on the comparison, it is shown that SOSJ is capable of better supporting the design and programming requirements of DISS that go beyond those in state of the art approaches. Also, SOSJ shows better overall performance in dynamic creation, suspension, resumption, and migration handling functionalities.
Acknowledgements

I would like to acknowledge and express my gratitude towards Professor Zoran Salcic and Dr. Kevin I-Kai Wang for their guidance, support, and wisdom throughout my PhD. Without their help, this ‘journey’ of PhD would not go this far and it is a great pleasure and honour to be one of their students.

I would also thank my father, Professor Siswanto Agus Wilopo, and mother, Ena Udayati for always supporting me (even though from afar). Massive love and gratitudes are due for my wife, Nur Anis Afifah Arzami, for her unconditional love and moral support during my final year of study. Great thanks to my older brother, Dayasakti Nawa Kartika and his family for temporarily hosting me whenever I needed to make temporary stops for my trip back home during holiday or a conference overseas.

Furthermore, I would also thank all the technicians (Mrs. Sunita Bhide, Mrs. Hoda Najafi, Mr. Rob Champion, and others) and the rest of the staffs at the Department of Electrical and Computer Engineering for their help and technical support. Many thanks also for my friends and colleagues, especially Dr. HeeJong Park for advices and supports regarding SystemJ-related issues, wherever they are for their countless help and assistance.

Finally, I would like to acknowledge the financial support of the University of Auckland through the University of Auckland Doctoral Scholarship. Without the scholarship, life could have been tougher for me.
# Table of Contents

Abstract ................................................................................................................................. i
Acknowledgements .................................................................................................................. iii
Table of Contents ..................................................................................................................... iv
List of Figures ........................................................................................................................... viii
List of Tables ............................................................................................................................ viii

1 Introduction .......................................................................................................................... 1
  1.1 Dynamic Intelligent Software Systems: Design Challenges ........................................ 2
  1.2 Motivating Example ....................................................................................................... 4
  1.3 Identified Design and Programming Requirements .................................................... 11
  1.4 Thesis Contributions .................................................................................................... 14
  1.5 Thesis Outline .............................................................................................................. 16

2 Background and Related Works ............................................................................................ 17
  2.1 Multi Agent Systems ..................................................................................................... 17
      2.1.1 JADE ..................................................................................................................... 17
      2.1.2 JIAC ..................................................................................................................... 18
      2.1.3 JADEX ............................................................................................................... 18
      2.1.4 JACK ................................................................................................................... 18
      2.1.5 Agent Factory ..................................................................................................... 19
  2.2 Service Oriented Architecture ....................................................................................... 19
      2.2.1 OSGI ................................................................................................................... 21
      2.2.2 UPnP ................................................................................................................... 21
      2.2.3 DPWS ................................................................................................................ 22
  2.3 Industrial Standard Approaches .................................................................................... 23
      2.3.1 IEC 61131 .......................................................................................................... 23
      2.3.2 IEC 61499 ......................................................................................................... 23
  2.4 Formal Model of Computation-based Approaches ....................................................... 24
      2.4.1 Esterel ................................................................................................................. 24
      2.4.2 SystemJ ............................................................................................................. 25
      2.4.3 DSystemJ .......................................................................................................... 27
      2.4.4 LibGALS and LibDGALS ................................................................................... 27
  2.5 Combination of Different Software Paradigms ............................................................. 28
      2.5.1 Holonic and MAS .............................................................................................. 28
      2.5.2 SOA and MAS ................................................................................................... 28
2.5.3 SOA and Industrial Standard Approaches ........................................ 29
2.5.4 Formal MoC and Industrial Standard Approaches ........................ 29
2.6 Qualitative Comparisons of State of the Art ...................................... 30
  2.6.1 Functional Correctness .................................................................. 30
  2.6.2 Dynamicity ................................................................................. 30
  2.6.3 Concurrency ................................................................................ 33
  2.6.4 Composability ............................................................................ 33
  2.6.5 Abstraction ................................................................................. 34
  2.6.6 Reactivity ..................................................................................... 35
  2.6.7 Data-Driven Computation Capability ........................................... 36
2.7 Summary and Conclusions ................................................................. 37
3 SOSJ : The Integration of SOA Paradigm with SystemJ .......................... 47
  3.1 Software Behaviours from the perspective of SOA and SystemJ (SOSJ) 48
  3.2 Service Description and Service Registry ........................................ 50
  3.3 SOA Functionalities ......................................................................... 56
    3.3.1 Beacon ..................................................................................... 57
    3.3.2 Advertisement ........................................................................... 58
    3.3.3 Request for Advertisement ......................................................... 60
    3.3.4 Discovery ................................................................................. 61
    3.3.5 Discovery Reply ........................................................................ 62
    3.3.6 Notify ........................................................................................ 63
  3.4 Service Invocation ............................................................................ 64
  3.5 Concluding Remarks ........................................................................ 69
4 Extending SOSJ for Dynamic Behaviours ............................................. 71
  4.1 Introducing CD Macro-states to SOSJ ............................................. 72
  4.2 CD and Subsystem Configuration in SOSJ ....................................... 76
  4.3 CD Macro-state Transitions Due to Dynamic Behaviours .................. 83
    4.3.1 CD Creation ............................................................................. 83
    4.3.2 CD Suspension ......................................................................... 85
    4.3.3 CD Resumption ......................................................................... 85
    4.3.4 CD Termination ......................................................................... 86
    4.3.5 CD Migration ............................................................................ 87
  4.4 Concluding Remarks ......................................................................... 89
5 SOA Implementation in SOSJ ................................................................. 90
  5.1 SOSJ Global Service Registry ............................................................ 90
    5.1.1 Design and Architecture ........................................................... 90
    5.1.2 Implementation ......................................................................... 92
  5.2 SOA RTS ......................................................................................... 99
A.2 Classes in the Original SystemJ RTS ................................................................. 214
A.2.1 SystemJRunner ................................................................................................. 215
A.2.2 SystemJProgram ............................................................................................... 216
A.2.3 JDOMParser ........................................................................................................ 218
A.2.4 Signal ................................................................................................................... 224
A.2.5 ClockDomain ....................................................................................................... 225
A.2.6 GenericSignalSender ......................................................................................... 226
A.2.7 GenericSignalReceiver ...................................................................................... 226
A.2.8 BaseInterface ..................................................................................................... 227
A.2.9 InterfaceManager ............................................................................................... 228
A.2.10 LinkQueue .......................................................................................................... 231
A.2.11 Interconnection and Link .................................................................................. 232
A.2.12 GenericChannel ............................................................................................... 232
A.2.13 input_Channel ................................................................................................... 233
A.2.14 output_Channel ............................................................................................... 236
A.2.15 Scheduler .......................................................................................................... 239
A.2.16 CyclicScheduler ............................................................................................. 239
A.2.17 GenericInterface ............................................................................................ 240
A.2.18 TCPIPInterface ............................................................................................... 240
B TransmitDisc Class .................................................................................................. 243
C ReceiveDisc Class .................................................................................................... 245
D User-Side Reconfiguration ....................................................................................... 248
E List of Publications .................................................................................................. 250
List of Tables

Table 2.1. SystemJ synchronous kernel statements and their descriptions .................................................. 26
Table 2.2. SystemJ asynchronous kernel statements and their descriptions ............................................. 27
Table 2.3. Comparison between existing design and programming approaches in terms of the identified design and programming requirements (except dynamicity) ................................................. 39
Table 2.4. Comparison between existing design and programming approaches in terms of the dynamicity requirement .............................................................................................................. 44
Table 5.1. SOSJ SOA function calls ........................................................................................................... 104
Table 6.1. SOSJ new function calls to trigger dynamic behaviours. ......................................................... 115
Table 7.1. The capability of SOSJ, JADE, WS4D JMEDS in satisfying the design and programming requirements .................................................................................................................. 159
Table 7.2. Details on the Conveyor, Diverter, and Loader CD ................................................................. 172
Table 7.3. Comparison between SOSJ and JADE in framework size ..................................................... 178
Table D.1. SOSJ GUI client application field names and their description .............................................. 249
List of Figures

Figure 1.1. An integrated ‘smart’ manufacturing, office, warehouse, and retail outlet facilities. .................................................. 5
Figure 2.1. SOA block diagram and the interactions between components. ................................................................. 20
Figure 2.2. Graphical illustration of an example of a SystemJ program.............................................................. 26
Figure 3.1. Illustration showing examples of scenario 1), 2), 3), and composite service. ..................... 49
Figure 3.2. Conceptual sequence diagram of the proposed SOA protocols which realize Publish and Find......................................................................................... 57
Figure 3.3. Channel reconfiguration for service invocation rule. ................................................................. 68
Figure 3.4. Timing diagram which pinpoints when reconfiguration occurs. ................................................. 69
Figure 4.1. All CD macro states and their transitions. ................................................................. 73
Figure 4.2. CD macro-state transitions due to dynamic creation of CD (creating new CD). .............. 84
Figure 4.3. CD macro-state transitions due to dynamic creation of CD (modify, update existing CD)............................................................................................................................................ 84
Figure 4.4. CD macro-state transitions due to dynamic suspension of CD. ......................................... 85
Figure 4.5. CD macro-state transitions due to dynamic resumption of CD. ......................................... 86
Figure 4.6. CD macro-state transitions due to dynamic termination of CD. ......................................... 86
Figure 4.7. CD macro-state transitions due to dynamic migration of CD at the source subsystem. ......................................................................................................................................................... 87
Figure 4.8. CD macro-state transitions due to dynamic migration of CD at the destination subsystem (strong migration). ........................................................................................................................................................................ 88
Figure 4.9. CD macro-state transitions due to dynamic migration of CD at the destination subsystem (weak migration). ........................................................................................................................................................................ 89
Figure 5.1. SOSJ Global Service Registry architecture. ................................................................................. 91
Figure 5.2. The process flowchart of the RegDiscReceiver thread ....................................................... 96
Figure 5.3. The process flowchart of the RegRecei ver class. ................................................................. 97
Figure 5.4. The RegSender thread process flowchart. ........................................................................... 98
Figure 5.5. Complete SOSJ RTS block diagram. .................................................................................. 99
Figure 5.6. The architecture of SOA RTS .............................................................................................. 100
Figure 5.7. The SOSJSOAR eceiver thread process flowchart............................................................ 102
Figure 5.8. The SOSJSOASender process flowchart .......................................................................... 103
Figure 5.9. Graphical illustration of the ‘template’ of consumer and provider CD. ....................................... 108
Figure 6.1. The Dynamic RTS as part of the SOSJ RTS (in red colour). ....................................................... 114
Figure 6.2. Sequence diagram showing the interaction of the LinkCreationSender and the MigrationAndLinkReqMsgThread. ................................................................. 117
Figure 6.3. Sequence diagram of link creation handshake........................................ 119
Figure 6.4. Sequence diagram of channel configuration for service invocation via channel. .... 122
Figure 6.5. The ReconfigInvServChannel() sequence diagram. .................................. 125
Figure 6.6. CD termination sequence diagram................................................................. 128
Figure 6.7. The process flowchart governing the functionality that handles CD termination. ... 129
Figure 6.8. CD creation sequence diagram. ..................................................................... 132
Figure 6.9. The process flowchart of the functionality denoted by the RunCreate() .......... 134
Figure 6.10. The “Instantiate Each Input (or Output) Channel & Link (if needed)” blocks...... 136
Figure 6.11. The “Check & Create Link (if needed)” block. ............................................. 137
Figure 6.12. CD suspension sequence diagram.................................................................. 139
Figure 6.13. The disableCD() method process flowchart within the RunSuspend() .......... 140
Figure 6.14. CD resumption sequence diagram............................................................... 142
Figure 6.15. The resumeCD() method process flowchart as a part of the RunResume() ....... 143
Figure 6.16. CD Migration sequence diagram.................................................................... 144
Figure 6.17. The RunMigration() process flowchart (top) and sequence diagram detailing the Initiate Transfer Functionality block of the MigHandshake() with the MigrationAndLinkReqMsgThread of the destination subsystem (bottom). ................. 147
Figure 6.18. The MigTransferThread and MigTransferRecThread interaction diagram....... 149
Figure 6.19. The reconfiguration of partner channels of the migrating CD’s channels in the same subsystem. ...................................................................................... 152
Figure 6.20. The initialization of the migrating CD in the destination subsystem. ............... 154
Figure 6.21. Migrated CD’s channels instantiation in the destination subsystem. ............... 155
Figure 7.1. Layout of the Discovery – Discovery Reply experimental setup ....................... 162
Figure 7.2. SOSJ Discovery – Discovery Reply Avg RT Time with different Advertisement expiry time .................................................................................................................. 162
Figure 7.3. Layout of the Request for Advertisement – Advertisement experimental setup..... 163
Figure 7.4. SOSJ Request for Advertisement - Advertisement average round trip time with different number of CDs and Advertisement expiry time......................................................... 164
Figure 7.5. Layout of the dynamic link creation experiment setup..................................... 166
Figure 7.6. Average total time to perform link creation process in different platforms. ......... 166
Figure 7.7. Average total time required to perform channel reconfigurations in case of single subsystem. ......................................................................................... 168
Figure 7.8. Average total time required to perform channel reconfigurations in case of distributed subsystems .......................................................... 168
Figure 7.9. The physical layout of the manufacturing example used for benchmarks. ............. 170
Figure 7.10. Average time to perform CD creation and termination to achieve Scenario 1 and 4. .................................................................................................................. 173
Figure 7.11. Average time for CD “update” to achieve Scenario 2 ........................................ 174
Figure 7.12. Average time for CD suspension and resumption to achieve Scenario 3 .......... 175
Figure 7.13. Average time for strong and weak CD migration to achieve Scenario 5 .......... 176
Figure 7.14. The layout of the Discovery/Probe – Discovery Reply/Probe Match experiment setup .................................................................................................................. 179
Figure 7.15. Comparison of Avg RT Time of WS4D Probe – Probe Match and SOSJ Discovery – Discovery Reply ........................................................................................................ 180
Figure 7.16. The average total round trip time of service invocations/two-way communications (single machine setting) in SystemJ, SOSJ, and WS4D JMEDS ................................................................. 182
Figure 7.17. The average total round trip time of service invocations/two-way communications (distributed machines setting) in SystemJ, SOSJ, and WS4D JMEDS ................................................................. 183
Figure 7.18. The average time comparison in performing behaviour creation & termination to achieve Scenario 1 & 4 – SOSJ & JADE ........................................................................................................ 185
Figure 7.19. The average time comparison in performing behaviour creation & termination to achieve Scenario 2 – SOSJ & JADE ........................................................................................................ 186
Figure 7.20. The average time comparison in performing behaviour suspension & resumption to achieve Scenario 3 – SOSJ & JADE ........................................................................................................ 187
Figure 7.21. The average time comparison in performing behaviour migration to achieve Scenario 5 – SOSJ & JADE ........................................................................................................ 189
Figure A.1. Diagram of Java classes in the existing SystemJ RTS and their relationships ..... 215
Figure D.1. GUI of the SOSJ client application ............................................................................. 248
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 3.3, Section 5.2.1, Section 5.2.2, Section 5.2.3, Section 7.1, Section 7.4.2, Section 7.4.3 of the thesis


Nature of contribution by PhD candidate

| Ideas, programming framework design & implementation, designed & ran experiments, analyzed results and performed qualitative & quantitative comparisons, manuscript preparation |

Extent of contribution by PhD candidate (%)

| 00 |

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salicic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</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
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salicic</td>
<td>![Signature]</td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>![Signature]</td>
<td>24/02/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 1.1, Section 1.3, Section 2.1, Section 2.4, Section 2.6 of the thesis


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Ideas, performed literature review, designed &amp; implemented the case study, performed analysis, performed qualitative &amp; quantitative comparisons, manuscript preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>85</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>HeeJong Park</td>
<td>Technical support &amp; advices</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
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Ž. Salcic</td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td></td>
<td>21/02/2017</td>
</tr>
<tr>
<td>HeeJong Park</td>
<td></td>
<td>21/02/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 2.6, Section 6.1, Section 6.2, Section 7.4.4 of the thesis


DOI: 10.1109/INDIN.2016.7819249

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Ideas, programming framework design and implementation, manuscript preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>90</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</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
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Z. Salcic</td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>T. C. F.</td>
<td>21/02/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 2.6, Section 3.3, Section 3.4, Section 3.2, Section 5.1, Section 7.2.1 of the thesis


| Nature of contribution by PhD candidate | ideas, SOSJ framework design and implementation, manuscript preparation |
| Extent of contribution by PhD candidate (%) | 90 |

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</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
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td></td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td></td>
<td>21/02/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 1.1, Section 1.3, Section 2.1, Section 2.4, Section 2.6 of the thesis


DOI: 10.1109/ParComW.2013.6529571

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Ideas, performed literature review, designed &amp; implemented the case study, performed analysis, performed qualitative &amp; quantitative comparisons, manuscript preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CO-AUTHORS</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</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
- the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>[Signature]</td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>[Signature]</td>
<td>21/02/2017</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
Co-Authorship Form

This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Chapter 4, Section 6.1, Section 6.2 of the thesis


doi: 10.1109/SOCA.2015.20

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Ideas, programming framework design and implementation, manuscript preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>90</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></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
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoran Salcic</td>
<td>[Signature]</td>
<td>24/02/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>[Signature]</td>
<td>21/02/17</td>
</tr>
</tbody>
</table>

*Last updated: 19 October 2015*
This form is to accompany the submission of any PhD that contains 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. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

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.

Content of the work is used in Section 2.1, Section 2.2, Section 2.4.2, Section 2.7, Section 3.3.2-Section 3.3.5 of the thesis


DOI: 10.1109/ICIEA.2015.7334250

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Ideas, SOSJ framework implementation, manuscript preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>90</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoren Salicic</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>Feedbacks and comments, manuscript preparation</td>
</tr>
<tr>
<td></td>
<td></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
- the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoren Salicic</td>
<td>📝</td>
<td>24/2/17</td>
</tr>
<tr>
<td>Kevin I-Kai Wang</td>
<td>📝</td>
<td>21/10/12/17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
Computing technology has a significant role in the history of modern industrial revolution. The industrial revolution has benefited from the utilization of computing technology, which presently moves towards what is known as the fourth industrial revolution or Industry 4.0 [1, 2]. Ever since the computing technology arrives at the domain of industrial manufacturing, the domain adopts the use of computing machines, typically in the form of Programmable Logic Controllers (PLC) [3], attached with industrial sensors and actuators to control and perform physical processes necessary to manufacture products. On the other hand, the recent advance in computing technology also enables computers to be smaller in size while also having significant computing capabilities. It opens the possibility where computers can be embedded into objects engaged in daily activities, therefore making them ‘smart’ and able to communicate with each other by means of wired/wireless communication networks. These objects, referred to as ‘smart objects’ [4], are not only embedded in the environment or even carried by or in possession of a human, but they may also be hidden from plain sight and integrated with daily activities without having anyone being aware of their existence [5, 6]. These smart objects are typically based on embedded computers or Sensor and Actuator Nodes (SANs) which are capable of sensing and actuating to their immediate environment and communicating with each other via wired/wireless network. These computing resources become more widely available as the computational technologies advance.

Industrial manufacturing process typically involves multiple different manufacturing operations handled by industrial machines with their intelligence implemented by software behaviours running on distributed PLCs or other types of physical controllers. Industrial machines are orchestrated to achieve proper operations to manufacture products. The orchestration of industrial machines is realized through the interaction between physical controllers or PLCs which are typically interconnected via industrial-typed networks such as EtherCAT [7], CAN [8], or ModBus [9]. With the increasingly pervasive smart objects, industrial manufacturing systems can be further enhanced with SANs, distributed and interconnected with each other via traditional networks (IP, Bluetooth, etc.) to perform sensing of ambient physical properties (e.g. temperature,
humidity, light intensity) and provide interfaces to control embedded devices and appliances [10, 11]. Similar with industrial machines, the behaviours of SANs are completely controlled by their corresponding software behaviours which provide them with certain level of intelligence. During runtime, industrial manufacturing systems may need to dynamically change to satisfy the continuously changing demand in production. These dynamic changes include the introduction of new individual machines/SANs, removal of existing machines/SANs, and reconfiguration of (some parts of) the existing machines/SANs. Such distributed systems, which can dynamically change and have concurrent software behaviours and provide certain level of intelligence to govern the behaviour of physical elements (e.g. industrial machines and SANs) is referred to as dynamic intelligent software systems (DISS) in this thesis.

1.1 Dynamic Intelligent Software Systems: Design Challenges

As previously mentioned, the increasingly pervasive computational resources allow traditional DISS to be enhanced with the use of smart objects (which are often based on embedded computers and SANs), deployed and distributed across the physical environment. Consequently, DISS encapsulates both the domain of Cyber Physical Systems (CPS) [13] and Ambient Intelligence (AmI) [14], with both being used in many real life application domains, spanning from residential homes [15, 16], healthcare centres/hospitals [17, 18], offices [19, 20], military [21, 22], transportation [23, 24], aerospace [25-27], nature and wildlife preservation [28, 29], search and rescue [30, 31], agriculture [32, 33], and industrial manufacturing [34-36].

Depending on their applications, DISS may use SANs for sensing and actuation purposes. While SANs are typically fixed and remain at their physical locations at all times (stationary), recent trend shows the use of ‘mobile’ SANs [37, 38], e.g. servicing robotics or human carried devices. Compared to stationary SANs which often have limited view angle and can be blocked by obstacles [39, 40], mobile SANs potentially give additional advantages by providing better sensing coverage and cover areas which are unreachable by stationary SANs. In addition, mobile SANs may also provide servicing and maintenance function to the corresponding system (e.g. replacing batteries of SANs) [41], especially in a condition when human can’t be involved due to the hazard or the difficult-to-reach locations where only mobile robots can reach/access. The recent advance in robotic technology leads to increasingly pervasive use of ‘aerial drones’ [42]. It allows the use of aerial drones as another type of mobile SANs with additional degree of freedom in their

---

1 Instead of being used to describe the type of systems like the ones in [12].
Introduction

field of view and movement (due to their aerial-based nature) compared to typical mobile robots and SANs which are terrestrial-based. Just like the terrestrial-based mobile SANs (or terrestrial SANs), these aerial drones can also be equipped with sensors and actuators, therefore becoming aerial-based mobile SANs, which potentially provide additional sensing coverage compared to terrestrial SANs and are capable of performing sensing and actuation in certain conditions or locations where human and even terrestrial SANs can’t access.

Also, the growing trend of the Internet of Things (IoT) [43] paradigm gives a distinct new feature for individual SANs to be IP-addressable. The IoT paradigm has entered different real-life application domains [44, 45], including the industrial manufacturing domain where it gives birth to the Industrial Internet of Things (IIoT) [46]. The IP-addressability of SANs allows the typically isolated SANs to be internet-accessible, therefore facilitating geographical location-agnostic interoperation through the Internet. The inclusion of (stationary and mobile) SANs and the IoT paradigm potentially extends the system scale, complexity, and types of man-machine interactions in CPS [38]. Naturally, the design, modelling, and implementation process of DISS are becoming more difficult without well-defined methodologies and software tools. As a result, system complexity is limited and the functional behaviour of the DISS is difficult to guarantee.

Depending on how software controllers are implemented on their execution platforms, software behaviours of DISS can be considered as concurrent programs running on centralized or (most often) distributed platforms. There are a number of design and implementation challenges in developing DISS [38]. First, DISS are considered as distributed systems which consist of a number of distributed computing machines or SANs operating asynchronously to each other, while also having a collection of (potentially asynchronous and synchronous) software behaviours running on individual machines. Both asynchrony and synchrony in DISS need to be properly addressed in order to ensure correct software system operation. Second, DISS can be highly heterogeneous. It may consist of different types of embedded computers and SANs with different computation capabilities, different types of sensors and actuators with varying physical input and output (I/O) interfaces and communication links. A higher-level of abstraction should be supported by the design and programming tools to ease design complexity. Third, each individual physical controller or SAN is considered reactive, i.e., it should be capable of responding to multiple correlated events coming from various sensing, actuating, processing and communication units. Design tools should properly support descriptions of reactive behaviours and ensure deterministic functions. This is important especially in DISS like industrial manufacturing system which can have stricter design and operational requirements. Manufacturing operations in industrial manufacturing systems have to be performed correctly with little room for errors/bugs. Therefore,
Introduction

Each software behaviour ideally should be verifiable for its functional correctness during the design time to facilitate the reliability of the manufacturing operations. Fourth, dynamic orchestration of software behaviours, ability to adapt behaviour for new configurations or even support dynamic creation and termination of behaviours are the key features of modern DISS due to the constantly changing presence of software behaviours following the modifications of the physical world and the mobility of SANs. These key features will also include the ability of (at least subsets of) software behaviours to migrate between computing machines or SANs.

1.2 Motivating Example

To better illustrate the challenges in designing and programming DISS, this section presents a motivating example of an integrated factory floor, warehouse, office, and retail outlet facilities. This example is presented to demonstrate sensing, actuation, collaboration between elements and human in a DISS built on top of multiple heterogeneous platforms of embedded computers and SANs following the vision of modern CPS and AmI paradigm.

Consider a scenario of a company which owns different facilities to support their production. It has office facility where most of the company’s employees and staff are based and perform their desk or office-related jobs. The company also has manufacturing facilities where products are manufactured and packaged. There is also central warehouse facility where manufactured and packaged products are stored temporarily prior to distribution to retail outlets. While the manufacturing facilities may store some quantity of raw materials, the central warehouse stores the majority of raw materials for stock. Finally, the company owns retail outlet which receives products delivered from the central warehouse and sells the manufactured products to the customers. It is worth noting that the company may own more than one of each type of the facilities.

To achieve safety and production requirements, the company decides to employ state of the art technology which uses SANs, embedded controllers, and robots in the facilities. The office facility has a stationary SAN S1, equipped with a proximity detector, to detect human presence. S1 also controls lighting and air conditioning system (based on detected human presence) in the corridor. There are two other stationary SANs S2 and S3 in the office, equipped with proximity detectors and cameras, to detect human presence and monitor the surrounding environment for security purposes. S2 and S3 also control the lighting and air conditioning systems in the office based on the detected human presence. Apart from the stationary SANs, there is also a terrestrial (mobile) SAN TS1 equipped with presence sensor which moves around the office or corridor performing security checks. The company also employs a service robot SR2 to perform tedious
Introduction

maintenance jobs (e.g. removing faulty stationary SANs, attaching new stationary SANs, replacing the batteries of stationary/mobile SANs, etc.). Service robots may also assist company staffs in performing domestic-related tasks [47] relevant to the office environment (e.g. serving coffee, fetching documents-mails). In addition, in the context of office-typed of environment, service robots can also be used for tele-presence purposes, allowing employees/staffs which are physically located in remote location to participate in activities occurring in the robot’s location to certain extent [48, 49].

Figure 1.1. An integrated ‘smart’ manufacturing, office, warehouse, and retail outlet facilities.
Introduction

The central warehouse facility has stricter safety and security requirements than the retail outlet facility since it stores larger quantity of manufactured products (compared to the local warehouse) which will be delivered to the (one or more) retail outlet facility. The products themselves may also require certain storage conditions as a quality control requirement to avoid the quality of the stored products from degrading while being stored, for example, the products have to be stored in certain ambient humidity or temperature, hence, it has to be satisfied by the central warehouse facility.

The central warehouse facility has four stationary SAN S8, S9, S10, S11 equipped with proximity and fire sensors to detect human presence and fire, respectively. They also control lighting system and air conditioning system to maintain the ambient physical condition according to the product storage condition requirements. Cameras are also attached to the stationary SANs in the central warehouse facility to monitor the surrounding environment for security purpose. Apart from stationary SANs, the central warehouse has terrestrial SANs TS4 and TS5 that patrol around the central warehouse premise to perform security checks. TS4 and TS5 have fire sensors attached on-board to detect any occurrence of fire when they navigate through the environment. Forklifts (FL1 and FL2) are also present, moving products from inbound shipping trucks that transport products from the manufacturing facility (for storage) and outbound shipping trucks that distribute products from the central warehouse to the retail outlet facility. It is worth noting that, different from traditional human-operated forklifts, the forklifts used in the facility are ‘intelligent’ and have built SANs and embedded controllers which provide the feature of real time localization and drive-by-wire capability [50], minimizing the operational responsibility of human operator in operating the forklifts. Such technology has been considered for industrial domain such as [51]. With this feature, human operator only operates forklifts during certain cases, e.g., when the drive-by-wire feature cannot be used or when forklifts are used to stack products on top of each other during storage. The shipping trucks also have built-in SANs and embedded controllers to introduce real time localization (on top of the use of GPS) the drive-by-wire feature which is activated when the shipping trucks are in the vicinity of either facility and ‘join’ the system.

The company also employs new state of the art aerial SANs AS1 equipped with a camera to navigate around the environment, performing security checks. AS1 may also be equipped with a Radio Frequency Identification (RFID) [52] or barcode scanner which allows it to identify products for inventory-checking purpose. It may also detect certain hazards such as fires when equipped with a flame/fire sensor. Besides inventory management and security checking, the company employs aerial SANs AS2 and AS6 to carry and deliver products from the central warehouse facility directly to the consumers. Note that, albeit still in its early phase, such product
delivery mechanism has been employed by well-known companies such as Amazon, UPS, and Google [53-55]. Similar to the shipping trucks that have ‘drive-by-wire’ feature, the aerial SANs movements are typically manually controlled by human when it is far away from the warehouse facility, and its ‘fly-by-wire’ feature [56] gets activated when it is in the vicinity of the warehouse facility.

Similar to the central warehouse facility, the manufacturing facility also has the same or even stricter level of safety, quality control, and security requirements due to the fact that manufacturing of products occurs in the manufacturing facility. In fact, compared to the central warehouse facility, the manufacturing facility possesses wider variety of potential hazards which can be associated with the manufacturing operations, e.g., hazardous liquid spills, flammable materials, gas leaks, explosions, and other hazardous risks. The manufacturing facility has four stationary SANs S4, S5, S6, and S7 which sense the ambient temperature and humidity of the manufacturing environment and control air conditioning and lighting system to maintain the ambient physical conditions at desired level following the manufacturing operational requirement. The terrestrial SANs TS2 and TS3 navigate through the environment to perform security check and detect undesired events caused by the aforementioned hazards (e.g. fire, dangerous liquid spill, gas leak, etc.). There is also an aerial SAN AS5 which is deployed outside of the manufacturing facility building, patrolling around the facility’s perimeter, detecting potential security breach.

Apart from having SANs, the manufacturing facility has multiple industrial stations, performing bottle capping and storage, controlled by embedded controllers. The overall manufacturing operation begins from bottle (filled with, e.g., carbonated drinks, honey, chemical liquids, etc.) getting loaded onto the conveyor belt either by a Bottle Loader station in point B, or point A if the Bottle Loader in point B is not operational due to maintenance or failure. The bottle is then transported to C, then to F via CB4, got capped by a capping station in G, and then stored by a storing station in point H before being packaged. A master controller MC allows human operator to monitor the status and condition of the manufacturing stations. Forklift FL4 is deployed in the manufacturing facility to transport products which have been packaged towards the shipping trucks for delivery to the central warehouse facility.

The retail outlet facility consists of two sections, the store and the local warehouse section. The store section has three stationary SANs S12, S13, and S14 equipped with camera to monitor the surrounding environment for security and surveillance purposes. They also control the lighting and air conditioning systems in the store section. The store section has customers which come to the store to buy product, therefore it has a distinct man-machine interface requirement which can’t
be offered by mobile SAN. Instead of having mobile SAN, the store section employs a service robot SR1, apart from only typical human shop attendants to assist customers, which has different, more attractive and human-friendly, appearance and man-machine capabilities than mobile SANs. Not only the service robot SR1 has a state of the art speech recognition and speech synthesis functionality that enables it to interact with customers, the service robot SR1 has a built-in SAN for ambient physical condition monitoring and real-time localization purposes. Since the local warehouse section stores products delivered from the central warehouse, it has a stricter security requirement than the store section. Apart from having scheduled visits by authorized employees or human security personnel for security checks, the local warehouse section has one stationary SAN S15 attached with proximity sensor and camera to monitor the surrounding environment for security purpose. The local warehouse section also has a terrestrial SAN TS6 which patrols around the local warehouse section, allowing improved security checks inside the premise. The retail outlet facility may also employ some forklifts, e.g. forklift FL3, to disembark delivered products from incoming shipping trucks to be stored in the local warehouse section. The retail outlet facility has aerial SANs AS3 and AS4 which, besides being able to use their on-board cameras for surveillance and security checks, can deliver products from the retail outlet facility directly to consumers.

In addition, humans may also be present in all of the facilities, conducting regular checks on manufacturing stations, controlling some parts of the manufacturing facility (the lighting or air conditioning), or simply doing office-related activities at the office facility. Humans may carry SAN which monitors their movement and health condition and provides information regarding their location in real time.

At any point in time, the company may change some of their operational, safety, security, or manufacturing requirements. This change may consequently require introducing additional changes to their facility. For example, stricter security and safety requirements will require introducing additional stationary SANs with additional surveillance cameras, mobile SANs, or human SANs to increase the frequency and intensity of security patrols/checks. Changes in manufacturing requirements may lead to the introduction of additional industrial stations and conveyors, removal of existing industrial stations and conveyors, or modification of the industrial station settings and changing of configuration of physical interfacing and communication links. Changes in operational requirements may lead to, e.g., additional forklifts and shipping trucks SANs to increase the overall throughput of loading, unloading, and shipping of products.
Introduction

While the four facilities can be located in the same geographical location, they may be geographically separated. This scenario becomes possible following the advance in the IoT and cloud technology [57]. Using the IoT and cloud technology, the four facilities can be located in separate geographical locations and interoperate through the internet. Referring to the motivating example, each facility has a host gateway node H1-H4 which collects information from all SANs in each of the facility and then made the information available for access by the user in a remote location via the Internet. In addition to the remote monitoring, an active control is also possible through host gateway node, which can accept commands from remote users via the internet to adjust the stationary SANs settings or control the mobile SANs movements.

Due to the interoperation enabled by the IoT and cloud technology, the company have the option to have their facilities in different geographical locations to reduce operational cost. As an example, the company can have their manufacturing facility in China, their office facility and central warehouse in New Zealand and Canada, respectively, and the company may own a number of retail outlets located in different countries around the world.

The flexible interoperation between facilities enabled by IoT and cloud technology will open many new opportunities and advantages. For instance, authorized staff can monitor the status of other facilities such as the manufacturing or central warehouse facility remotely or through a control and monitoring interface in the office facility. From the supply chain point of view, the manufacturing facility may perform on-demand manufacturing of products if the central warehouse facility is low on stock and requires the manufacturing facility to manufacture additional products. Customers may also order the products with certain customizations on-demand e.g. fruit juices with different flavours, different ingredients, particular dietary requirements with certain nutritional content (e.g. low sugar, gluten-free), different bottle colour, different bottle cap colour, different bottle material, different bottle size to accommodate different volume of drink, etc., in which the request can be passed immediately to the manufacturing facility to manufacture. Alternatively, if the requested customized products are in stock in the central warehouse or available in other retail outlet facilities, it can be dispatched immediately to the requesting retail outlet or directly to the consumers from the closest facilities which have the requested product in stock.

From the managerial point of view, flexible interoperation with IoT and cloud technology will allow for more efficient resources sharing between facilities. For example, the company may own more than one facility of each type and some may be considered closely located geographically. One of the manufacturing facilities may be chosen to manufacture the product due to having most
of the required materials in stock, however it may require additional quantity of some required materials. The interoperation between facilities can allow, e.g. one manufacturing facility to request materials (required for production) from another manufacturing facility via the internet when the need arises. Another example, in case of emergency, one facility may request other facility to dispatch mobile SAN to the facility to, e.g., reinforce security. The features provided by the IoT and cloud technology can also be used by the company when they expand and add more company facilities, or when they have to ‘shrink’ and remove existing facility for efficiency. When the company has new facilities due to expansion, the new facilities can be integrated together and interoperate with the existing facilities using the IoT and cloud technology, regardless of the new facility’s geographical location. Similarly, when the company removes existing facility due to company shrinking, the existing facility being removed can detach from the other existing facilities, and the other existing facilities may continue their interaction and interoperation as it is.

While it is expected that the entire system will run normally, the possibility of failures can’t be ruled out. Some parts of the system of the motivating example may experience failures. Also, regular maintenance operations may require occasional suspension of some parts of the system. While some parts of the system may be stopped due to these conditions, the manufacturing facility should be able to continue the production process. For example, referring to Figure 1.1, during runtime the conveyor belt CB4 or the diverter D1 may be stopped for maintenance or due to failure. In this condition, the industrial robot IR1 can take over the operation to move bottle from point C to point F. If the robot R1 is also stopped for maintenance or due to failure, the conveyor CB2 can take over the transportation of bottle from C to F via point D and E, albeit with an increase in bottle travel distance. Besides temporary halt of the parts that perform industrial manufacturing, the stationary and mobile SANs may also stop operating due to maintenance or failure (e.g. battery replacement, exhausted battery). For instance, when one of the stationary SANs that performs temperature or humidity sensing fails, one of the mobile SANs may take over the sensing operation of the stationary SAN, if deemed necessary. With the geographical location-agnostic interoperation between facilities, the mobile SAN which is taking over the sensing operation of the stationary SAN may originate not only from the same facility of the stationary SAN, but also from different facilities.

The motivating example demonstrates a potentially large scale DISS that exhibits high-level of dynamic behaviours due to, e.g., the mobility of mobile and human SANs, and possesses highly heterogeneous elements that are capable of cooperating and collaborating with each other to achieve mutual goals. Each has different sensing, actuation, and communication capabilities, and exhibits high degree of reactivity implemented by distributed and concurrent software behaviours.
Introduction

The presented motivating example will be used to identify design and programming requirement of distributed intelligent software systems in the next section.

1.3 Identified Design and Programming Requirements

Based on the motivating example, a number of requirements or necessary features of the design and programming tools for describing and implementing distributed intelligent software systems is identified as follows [38, 58, 59]:

1) Functional Correctness

A robust DISS functionality can be difficult to achieve when programmers have to ensure and verify the correctness of the design and of every software behaviours in the system while also taking into consideration the distribution and concurrent nature of software behaviours. To ensure correct system functionality, a design and programming framework ideally should be based on formal semantics and comply with some Model of Computation (MoC) to facilitate correct-by-construction design and to avoid bugs and errors prior to system deployment. In addition, the designed software behaviours should be formally verifiable. Without such feature in the underlying design and programming tool, programmer typically have to resort to manual tests for identifying errors and bugs to ensure correct system functionality, which often require tremendous amount of time and resources. However, such exhaustive testing is generally not feasible for complex DISS, it becomes even more unfeasible when dynamic behaviours are involved and taken into consideration. In addition, such manual and exhaustive testing offers no guarantees that all possibly occurring errors/bugs are found during the test.

2) Dynamicity

According to [60], dynamicity can be defined as the ability to cope with dynamic behaviours and “uncertainties” at a system level. These uncertainties are caused by the change of presence of software behaviours, which is often associated with software behaviours entering and leaving the system at will, i.e., the addition of software behaviours to the system and removal of software behaviours from the system. Addition of software behaviours occurs at creation of new software behaviours, resumption of suspended software behaviours, and migrating software behaviours (when they arrive in the new location and then commences its execution). On the other hand, removal of software behaviours occurs at permanent termination of software behaviours, suspension of software behaviours execution, and also at migration of software behaviours to another location. Software behaviour migration is often classified into two types, strong migration and weak migration. Strong migration involves the mobility of software behaviour with its data
Introduction

and execution state retained for the software behaviour to resume its execution at the new location, whereas weak migration enables the mobility of software behaviour without its data and execution state retained, therefore making the software behaviour to restart its execution from the beginning in the new location.

The addition and removal of software behaviours are closely related with the issue of scalability and fault tolerance [60]. Also, having individual software behaviours to be aware of the presence of other software behaviours can be considered a major advantage for performing dynamic behaviours. In this thesis, the term ‘behaviour awareness’ will be used to describe this requirement.

- **Scalability:** Referring to the motivating example, the addition or removal of industrial stations to existing manufacturing lines and the addition and removal of SANs, both can be associated with addition and removal of software behaviours which lead to dynamic presence of software behaviours. Also, the integration and separation of facilities (facilitated via the internet) produce even higher degree of dynamic presence of behaviours.

- **Fault tolerance:** The hardware components of the system may be the root cause of any events of failures. Any hardware failure will correspond to one or more software failure due to software behaviours unavailability, which affects the overall system behaviour. For industrial manufacturing application, any failures affecting the overall production process can cause significant financial losses. Therefore, it is crucial for the programming paradigm to be able to handle such dynamic situations and try to maintain the system functionality, even at a reduced performance or throughput.

- **Behaviour awareness:** It’s desirable that individual software behaviours are aware of the presence and the capabilities of other software behaviours in the system to facilitate dynamic interaction and orchestration between individual software behaviours.

Also, dynamic behaviours will often require the reconfiguration of software behaviours including correct re-establishment of communication between existing behaviours and connections to the environment. Therefore, it is imperative that dynamic behaviours and reconfiguration should be supported by the underlying design and programming frameworks. In addition, the underlying design and programming frameworks have to provide sufficient support so dynamic behaviours and reconfiguration are performed without the need to restart the entire system and affecting other existing software behaviours in the system.
Introduction

3) Concurrency

Being able to handle concurrent software behaviours is one of the most important requirements for any design and programming framework for DISS. From the motivating example described in Section 1.2, it is clear that the scenario contains different levels of concurrency. For instance, the motivating example has a number of distributed stationary SANs, mobile SANs, human SANs, forklifts and shipping trucks equipped with SANs, and also embedded controllers governing the behaviour of industrial stations and machineries. All the aforementioned elements may work collaboratively and are running asynchronously with each other, while each of them may contain concurrent software behaviours which may run both asynchronously and synchronously to perform different types of operations in order to achieve specific goals. Without any proper concurrency handling, the system may become functionally incorrect, and even worse, concurrency issues such as deadlocks may cause the system to stop operating. Hence, the design and programming framework has to be able to provide programmers proper handling of asynchronous and synchronous concurrent software behaviours while avoiding situations such as deadlocks and race conditions.

4) Composability

DISS similar to the motivating example is typically subject to extensions and inclusion of additional embedded controllers or SANs to acquire more context information from the environment or to perform additional operations following changes in application requirements. Therefore, a programming framework should support a design paradigm that facilitates hierarchical composition, modularity, and code reusability to enable easy expansion of system functionalities. Moreover, a system designed via a modular approach allows easy modifications and adjustments in case of any desired or required changes.

5) Abstraction

As illustrated by the motivating example in Section 1.2, DISS typically contains different types of sensors, actuators, and computing platforms, causing DISS to be diverse and heterogeneous in nature. Without sufficient abstractions, programmers often have to resort to low-level implementation details, such as hardware resource access and sharing, communication protocols, and physical interfacing implementation which could lead to a more error-prone system implementation. Hence, it is necessary for a programming framework to provide sufficiently high-level of abstraction and hide low-level details to minimize burdens during the design process. With
Introduction

sufficient level of abstraction, designers can focus on implementing system-level functionalities rather than low-level control and communication.

6) Reactivity

Typical DISS are considered as reactive systems, where they process information acquired from sensors and actuate to the physical environment accordingly. In such systems, it is very likely that multiple correlated events come in different orders due to common synchronization issues. However, individual software behaviours in the system should behave in a deterministic manner, i.e. the same sequences of input events will lead into the same sequences of output actions. It can be a difficult and challenging task to manually implement reactive software behaviours, with guarantee that they will behave in a deterministic manner, without proper programming constructs and support from the underlying design and programming tools. Also, it is crucial for the underlying design and programming tools to support safe pre-emption in response to events with explicit scope and execution flow to ensure that individual software behaviours in the system remains deterministic and analysable.

7) Data-Driven Computation Capability

Apart from being reactive in nature, it is most typical for individual software behaviours of DISS, in particular the ones similar to the motivating example, to possess fairly complex data-driven computational component. For example, the software behaviours which govern the behaviours of service robots, industrial robots, manufacturing stations, and mobile SANs employ certain control algorithms such as the Proportional, Integral, and Differential (PID), neural network, or genetic algorithms which may require relatively complex data-driven computational elements. Service robots may also use speech recognition and synthesis functions which often involves complex data-driven computation. Hence, the underlying design and programming framework should provide and be capable of data-driven computation.

1.4 Thesis Contributions

This thesis focuses on proposing a novel design and programming framework which is capable of modelling, describing, and satisfying all of the aforementioned identified design and programming requirements of dynamic intelligent software systems. The main contributions of this thesis are:
Introduction

1) *A novel software programming paradigm for designing and programming DISS achieved through the synergy of service oriented architecture and a formal MoC-based system-level language SystemJ*

Based on the current research gaps which are identified by analysing the capability of the current state of the art of design and programming approaches in satisfying the design and programming requirements of DISS, the synergy between the SOA paradigm and a system-level language SystemJ [61] is considered to address the identified research gaps. This synergy leads to the novel programming approach named Service Oriented SystemJ (SOSJ). The novel approach makes use of the features of SOA in providing loose and dynamic coupling between software behaviours and its information exchange mechanisms with the features-laden of SystemJ for programming distributed systems to satisfy the design and programming requirements of DISS.

2) *The introduction of macro-states of clock domain*

The SystemJ CD (mutually asynchronous concurrent software behaviour containing mutually synchronous concurrent software behaviours)\(^2\) semantics define that once execution starts, they will always be run in every tick during runtime. To introduce dynamic behaviour handling mechanisms in SOSJ, the macro-states of CD and their transitions are defined, which extend the existing CD semantics to cater for dynamic behaviour scenarios and formally define and govern the full life cycle of CD in DISS during runtime.

3) *Descriptions of the SOSJ framework and mechanisms to handle full dynamic behaviours and reconfiguration.*

The proposed approach of synergizing SOA paradigm with system-level language SystemJ leads to the extension of the existing SystemJ with new mechanisms used by the programmers to describe dynamic systems and also new Runtime Support (RTS) to support and perform full dynamic behaviours and reconfiguration. These modifications of the RTS include the introduction of new Java threads to handle the transmission and receiving of SOA messages to enable information exchange mechanism as defined by the SOA paradigm, new data structures to implement and store service description needed in SOA, and new mechanisms and programming constructs which enable handling of dynamic behaviours. A graphical software tool is also developed to allow human users or programmers to trigger dynamic behaviour and reconfiguration from remote machines.

\(^2\) Refer to Section 2.4.2 for more detail
Introduction

4) **Qualitative comparisons between SOSJ and state of the art programming approaches, analysis, and performance evaluation of the SOSJ framework.**

The new design and programming framework SOSJ is qualitatively compared with the existing state of the art of design and programming approaches in satisfying the identified design and programming requirements of DISS. Several scenarios based on the motivating example are proposed as case studies for performance benchmarks to demonstrate the capability of SOSJ in handling and performing dynamic behaviours and reconfiguration.

1.5 **Thesis Outline**

The essential background information needed to understand the rest of the thesis and also the information regarding the available state of the art of design and programming tools for dynamic intelligent software systems are elaborated in Chapter 2. Next, Chapter 3 presents SOSJ as a novel design and programming framework created through the synergy of a software paradigm called Service Oriented Architecture (SOA) and a system-level language SystemJ. Chapter 4 describes SOSJ macro-states and their transitions which form SOSJ lifecycle. Chapter 5 provides the full description of the extension and modification of the SystemJ Runtime Support to introduce the SOA functionalities along with additional programming constructs and mechanisms to allow programmers to use them. Chapter 6 presents the full description of the extension and modification of the SystemJ Runtime Support to introduce the dynamic behaviour handling functionalities along with additional programming constructs and mechanisms to allow programmers to use them. Chapter 7 presents the qualitative comparison between SOSJ and the existing state of the art in satisfying the design and programming approaches of DISS, analysis, and performance evaluation of the SOSJ framework based on case studies derived from the motivating example in Section 1.2. Finally, Chapter 8 presents concluding remarks and potential future research works.
This chapter presents the state of the art of design and programming approaches amenable for DISS. These design and programming approaches, which include the Multi Agent Systems [62], Service Oriented Architecture [63], formal MoC-based approaches [64], industrial standard approaches [65], and combination of different approaches are investigated and compared based on the identified design and programming requirements of DISS elaborated in Section 1.3.

2.1 Multi Agent Systems

The Multi Agent Systems (MAS) are developed using agent frameworks. Agent frameworks aim to provide abstraction by encapsulating software behaviours into entities called agents, where each agent is loosely coupled with other agents and have some level of reactivity. In MAS, agents are executed concurrently and the MAS frameworks provides the facility for agents to communicate with each other via certain standardised interfaces. The following MAS frameworks, JADE [66], JIAC [67], JADEX [68], JACK [69], and Agent Factory [70] are discussed in the following sections.

2.1.1 JADE

JADE (Java Agent DEvelopment framework) is a software framework specially catered to allow programmers to develop agent-based applications. Originally developed by Telecom Italy Lab in collaboration with several academic and research institutions, JADE is based on Java programming language and complies with the FIPA (Foundations of Intelligent Physical Agents) standard [71]. JADE’s compliance with the FIPA standard allows JADE agents to communicate with other FIPA-compliant agents. Available within JADE framework are libraries to allow programmers to develop software agents and handle their behaviours and communications. Since its emergence, there have been libraries made available as add-on packages for JADE, for example, the Lightweight Extensible Agent Platform (LEAP) [72]. The LEAP library provides the support for creating agents and enables JADE agents to be deployed on smaller computing platforms. Apart from its agent runtime-environment, JADE also has built-in toolkit for the purpose of debugging.
and administration of agents. For instance, the JADE runtime environment has a Directory Facility agent which is in charge of providing a lookup function for all registered agents and a Sniffer agent to monitor agent communications.

2.1.2 JIAC

JIAC (Java Intelligent Agent Componentware) is a software framework which allows for the development of agent-based applications. It is originally based on Java language to allow deployment on heterogeneous platforms. JIAC was originally created for telecommunication purposes, however, it has undergone further development to allow targeting different applications including industrial manufacturing. Different than JADE, JIAC agents are based on the Belief, Desire, and Intention (BDI) model [73]. The first version of JIAC is made to specifically use Java, however the most recent version of JIAC called JIAC V [74] has been made compatible with other languages. One particular agent programming languages used by JIAC V is JIAC Agent Development Language ++ (JADL ++), the successor of JADL language [75], which allows programmers to describe ontology using the ontology-based language OWL [76]. JIAC V adopts the SOA paradigm so it can be considered to fall into a different category of design and programming approaches discussed in Section 2.5.2

2.1.3 JADEX

JADEX is a Java-based agent framework which is developed as an extension of the JADE agent framework. JADEX adds the BDI reasoning engine to enable the development of goal-oriented agents. JADEX possesses some similarities with JIAC as mutual BDI-based agent platforms. Since JADEX is an extension of JADE, JADEX agents also uses all of the features provided by JADE framework, which includes FIPA-compliant communication, FIPA-based agent management, and security features.

2.1.4 JACK

JACK, or also known as JACK Intelligent Agents, is a MAS framework proposed and developed by Agent Oriented Ltd. It is based on Java language and follows the BDI model. Apart from being able to use ordinary Java classes, JACK agents are described using a unique agent-oriented language named JACK Agent Language, which is extended from Java to program JACK agents. To develop JACK agents, programmers use the JACK development tool which comprise a compiler which compiles programs written in JACK Agent Language into Java programs, a graphical development environment called JACK Development Environment which is accompanied with a set of development library based on Java named Java Agent Kernel.
Background and Related Works

2.1.5 Agent Factory

Agent Factory (AF) is an open source programming framework which allows the development and deployment of agents. It is originally proposed and developed by University College Dublin. The framework uses FIPA-compliant architecture and a number of different agent programming languages which include AFAPL [77], AFAPL2 [78], AF-AgentSpeak which is extended from the AgentSpeak language [79], AF-TeleoReactive [80] which is extended from the Teleo Reactive agent language [81], and AF-ASTRA which is a combination of both the AF-AgentSpeak and AF-TeleoReactive language. The framework distribution comes with a built-in runtime environment and a generic development environment which is created as an Eclipse IDE extension accompanied with a compiler, and a set of programming libraries which facilitate the programming of agents in different aforementioned agent programming languages. Agent Factory agents can be developed using the aforementioned agent programming languages, which they can be compiled into Java and then executes on a range of execution platforms. The framework distribution has two versions, the Agent Factory Standard Edition (AFSE) [82] which targets more powerful computing machines (e.g. PC, laptop) and Agent Factory Micro Edition (AFME) [83] which is developed for less powerful computing machines.

2.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is an approach which promotes the concept of service as discrete functionality that form applications. In SOA, the behaviours of these discrete functionalities (services) are governed by service entities which implement them. The paradigm promotes the concept of loose coupling between service entities and offers flexibility in which services entities can be composed and orchestrated at runtime to form different applications. Service entity typically has the following three elements, service description, service interface, and implementation. Service description contains a set of information about the services offered by the corresponding service entity, including the capability of the service governed by the corresponding service entity and service interface. Service interface hides the complexity of service implementation and provides access points in which services can be accessed and invoked by clients. Service implementation describes how the service offered by the corresponding service entity performs its functionality to generate result.
Background and Related Works

![SOA Block Diagram](image)

Figure 2.1. SOA block diagram and the interactions between components.

Typically, each SOA-based application has or is based on the following three main components, service registry, service provider, and service consumer. The interactions between these three components are shown in Figure 2.1 (recreated from [63]). A service provider is a service entity which offers services that can be used by service consumer, while service consumer uses services offered by service providers. Service registry has repository which contains the descriptions of services offered by service providers. The common interactions between these components involves the following consecutive operations, publish, find, and bind [63]. Once service registry informs service provider of its presence, service provider advertises its services by sending its service description to service registry for service consumer to discover (Publish). Next, service consumer discovers services by making a query to the service registry and receiving reply from service registry containing a list of service descriptions stored in the service registry (Find). After receiving reply from service registry, service consumer may perform service matching to decide which services it will invoke from the list of service descriptions obtained from the service registry. Finally, service consumer can invoke ‘matching’ service via the defined interfaces to obtain result (Bind).

There are several key features in SOA that facilitate dynamicity in DISS, including:

- **Loose coupling:** Service entities are mutually independent and (should be) implemented in a way which minimizes dependencies with other service entities. The feature of loose-coupling allows for dynamic orchestration of service entities during runtime, which is necessary for developing highly dynamic and scalable systems [84].
- **Information exchange mechanism:** The interaction shown in Figure 2.1 provides the mechanisms for service entities to be informed of which and what services are advertised and can be invoked by service consumers dynamically.
Background and Related Works

- Flexibility: In SOA, applications are not necessarily constructed statically before runtime. Because service entities are loosely-coupled, they can bind with each other dynamically to form different applications during runtime. When the presence of service entities changes during runtime, the presence of services (offered by service entities) will consequently vary and thus service consumer has to find alternative services which offer similar functions with the unavailable services.

Originally found typically in business processes and enterprise-based systems, the SOA paradigm has captured the attention of those working in the domain of distributed systems. Some examples of rather widely known SOA-based software technologies, OSGi [85], UPnP [86], and DPWS [87], are discussed as follows.

### 2.2.1 OSGI

OSGi (Open Services Gateway initiative) is an open-standard software framework based on Java. OSGi abstracts software applications as ‘bundles’ with clearly defined interfaces. The framework itself runs on top of Java Virtual Machine (JVM) and provides a runtime environment which governs bundles’ lifecycle, i.e., allowing OSGi bundles to be installed, started, stopped, updated, and uninstalled without the need for system restart. The OSGi framework provide the functionality which allows OSGi bundles to publish, find, and bind services dynamically. Once an OSGi bundle is installed, it advertises its services to the service registry in the framework. Once advertised, they become discoverable by other bundles, in which then other bundles may use the advertised services. Any dependencies required by bundles are managed by the framework during runtime. Currently, the OSGi Alliance, which consists of many industrial and academia collaborators, is responsible for defining the OSGi standards and performing further development of the OSGi framework. There are also a number of SOA-based frameworks which are developed based on or extensions of OSGi, such as the MCC-OSGi [88], R-OSGi [89], OSGi4C [90], alfRedO [91], OSGi-PC [92], Sensor Node Plug-in System [93, 94], CBD-OSGi [95], AIOLOS [96, 97], A-OSGi [98], and Androsgi [99].

### 2.2.2 UPnP

UPnP (Universal Plug and Play) is a SOA-based specification which provides an architecture for peer to peer communication between devices. Originally developed by Microsoft and now by the UPnP Forum, the technology is built from the Plug and Play concept which is deployed into network-capable devices. The UPnP architecture uses several standard protocols including the Simple Service Discovery Protocol (SSDP) [100], HyperText Transfer Protocol (HTTP) [101], and Simple Object Access Protocol (SOAP) [102]. The UPnP architecture defines the concept of
Background and Related Works

two types of devices, controlled devices and control points. Controlled devices may provide one or more services and respond to requests sent from control points. Each controlled devices and control points in UPnP has to be assigned with an IP address. A UPnP network is established through the search and advertisement mechanisms implemented in SSDP. The first step is referred to as discovery. When a controlled device comes into the network, its services are advertised to control points on the network. Likewise, when a control point joins the network, it may search for controlled devices of interest on the network. Once the discovery step is finished, the next step is referred to as description. In this step, control points which have discovered controlled devices of interest may retrieve the description of controlled devices which describes about the controlled devices and their capabilities via the URL given in the discovery message. Then, the third step is referred to as control. During this step, the control point has learned about the controlled devices description and capability of interest and then it can start to send a command to controlled devices to retrieve results. During runtime, control points may also subscribe to notification (referred to as ‘event’ in UPnP terminology) when the service provided by controlled devices are updated or modified.

2.2.3 DPWS

Device Profile for Web Service (DPWS) is a SOA specification which aims to bring secure Web Service protocols into the hardware level. After submitted for standardization in 2008, the DPWS specification got standardized by Organization for the Advancement of Structured Information Standards (OASIS) in 2009. The protocol uses several standards and specifications typically used in web services, which include SOAP, XML Schema [103], Web Service Description Language (WSDL) [104], and also Web service-specific standards including WS-Discovery, WS-Addressing, WS-Eventing, WS-MetadataExchange, and WS-Policy. WS-Discovery allows services to discover other services in other devices and advertise themselves. WS-Addressing defines the procedure to provide unique address of devices (where the web services are running) in DPWS (in SOAP format) messages. WS-Eventing permits devices to subscribe to asynchronous messages from devices of interest. WS-MetadataExchange provides the mechanism for other devices to obtain the metadata that describes how they can contact and use the services provided by the device. WS-Policy allows describing a set of specifications regarding capabilities, constraints, and requirements of web service, e.g., security, Quality of Service (QoS), in XML. According to the DPWS specification, a device with services that joins DPWS network transmits an announcement to inform its presence ('hello' message). Next, clients may attempt to discover services in the network using certain search criteria, e.g., the name of the service or the service type (by transmitting ‘resolve’ or ‘probe’ message). After clients have found service of
Background and Related Works

interest, they may transmit an invocation message (or ‘operation invocation’ message in DPWS terminology) to the service to retrieve results. Any devices that wish to leave the network transmits an announcement (‘bye’ message) to inform all clients.

One of the most popular software programming toolkits which uses the DPWS protocol is the WS4D (Web Service for Devices) [105]. The WS4D contains the implementation of DPWS protocol and communication stacks developed in C, C++, and Java. WS4D comes as a result of the SIRENA research project which aimed to interconnect the industrial, telecommunication, automotive and home automation domain through SOA [106]. Other works which also use WS4D are SOCRADES [107] and IMC-AESOP [108] which also continue the development of DPWS stack in WS4D for embedded and DPWS-capable devices.

2.3 Industrial Standard Approaches

These approaches are specifically designed to comply with industrial manufacturing-typed applications which are typically based on Programmable Logic Controller (PLC). In the domain of industrial manufacturing, two of the most prominently used approaches are the IEC (International Electrotechnical Commission) 61131 [109] and the IEC 61499 [110]. Both approaches are described in the following.

2.3.1 IEC 61131

The IEC 61131 is deemed as the legacy approach and guideline for PLC programming. IEC 61131 was originally named IEC 1131 and developed by the working group SC65B WG7 of the International Electrotechnical Commission organization. Because of the involvement of different PLC manufacturers when the standard was first established, many PLC manufacturers design their PLCs to be compatible with the IEC 61131 standard, and therefore, the IEC 61131 become the mainstream approach and currently one of the most widely used in industrial manufacturing applications. The IEC 61131 is classified as a data driven approach with different types of PLC programming languages, ranging from the textual-typed Instruction List (IL) and Structured List (SL) to the graphical-typed Ladder Diagram (LD) and Function Block Diagram (FBD).

2.3.2 IEC 61499

The IEC 61499 is considered as the next successor of the IEC 61131. Different from the IEC 61131 which is data-driven, the IEC 61499 is event-driven and aimed at allowing a modular-based approach in designing modern manufacturing applications which are typically distributed in nature. The IEC 61499 proposes the use of function blocks as the basic building blocks for applications. Each function block encapsulates data, a set of Finite State Machine (FSM) which
Background and Related Works

govern the behaviour of function block, and IEC 61131 software behaviours. Individual function blocks are reusable and can be composed and orchestrated together via separate event-data interfaces to form a complete system. Similar to the IEC 61131, the IEC 61499 standard also possesses the object orientation capability. Despite it has been around since 2005, the standard hasn’t been adopted extensively and so far several IEC 61499-compliant programming tools and PLC vendors are available. Among the currently available IEC-61499-compliant tools are the Function Block Development Kit (FBDK) [111] with the accompanied runtime environment Function Block Run Time Environment (FBRT), IsaGRAF Workbench [112], NXTStudio [113], 4DIAC-IDE [114, 115], FBench [116] (which is originally from the OOONEIDA Workbench [117]), BlokIDE [118], and CORFU/Archimedes [119].

2.4 Formal Model of Computation-based Approaches

There have been attempts to develop distinct approaches which make use of formal semantics and Model of Computation (MoC) to allow designed software behaviours to be functionally correct and become amenable for formal verification and validation, reducing programmer’s efforts in software testing and identifying errors/bugs in their designs. Currently, a number of approaches which are based on formal MoC are available. Some of them, namely Esterel [120], SystemJ [61], DSystemJ [121], LibGALS [122], and LibDGALS [123], are elaborated in the following.

2.4.1 Esterel

Esterel is a programming language which is based on synchronous reactive (SR) MoC [124]. It is an imperative programming language with programming constructs to allow programmers to develop reactive systems. Esterel programs can be compiled into C code or hardware-specification code such as VHDL [125] or Verilog [126]. Different than conventional programming languages like Java or C, Esterel has a distinct notion of time such that software behaviours in Esterel (reactions) advance in lock-step according to logical discrete clock called tick. Due to having the underlying formal SR MoC, Esterel programs become amenable for formal verification and validation. Reactions in Esterel communicate with each other and with the environment through an abstracted mechanism called signal. Esterel signals can be of either two types, pure or valued signals. Pure signals have only boolean status, i.e., they can be either present or absent. Meanwhile, valued signals contain both boolean status and value. Once emitted, the signal’s status and value (if any) become visible to all reactions in the CD in the same tick. This is known as the instantaneous signal semantics.
2.4.2 SystemJ

SystemJ is a system-level design and programming language amenable for designing distributed systems. It is based on formal semantics and the Globally Asynchronous Locally Synchronous (GALS) MoC [127] which originally comes from the hardware domain. SystemJ also adopts programming constructs and reactive statements from Esterel for control-dominated operations and the object orientation features of Java for data-driven operations. Due to the underlying formal semantics and GALS MoC, software behaviours in SystemJ are also formally verifiable. SystemJ allows mutually synchronous behaviours, called reactions, to be grouped into mutually asynchronous behaviours called clock domains (CDs). Similar to Esterel, SystemJ also enforces the use of tick as the notion of time. However, while the synchronous model in Esterel is applied to one Esterel program, SystemJ limits the synchronous model within one CD and enforces the asynchronous model between different CDs.

In SystemJ, the signal mechanism is also used for reactions to interact with the environment and between each other in the same CDs. Once signal is emitted and becomes present, its status and value (if any) will be visible to all reactions in the CD. It is worth noting that, different from Esterel which complies with the instantaneous signal semantics, SystemJ employs a different signal semantics called delayed semantics [128]. This causes signal emission to be visible to all reactions in the same CD during the next tick, therefore completely preventing any issues which might emerge from cyclic signal dependency.

On the other hand, a different mechanism called channel is used for reactions in different CDs to interact with each other. Different from signal which can be used for one-to-many communication, SystemJ channel is strictly one-to-one (point-to-point) communication. SystemJ channel has the inherent feature of rendezvous adopted from the Communicating Sequential Processes (CSP) [129] which guarantee data delivery. It is worth noting that the CSP-based channel communication semantics in SystemJ enforce static or fixed coupling between sender and receiver. Communications performed through signals and channels don’t require the programmers to deal with low-level details of physical interfaces (e.g. sensor/actuator interfaces, network protocols). The channels and signals are mapped into various physical interfaces based on the provided CD configuration written in the XML format. The physical interfacing with the environment are handled by the SystemJ Runtime Support (RTS) implemented in Java. One or more CDs can be composed together to form a SystemJ subsystem (SS), in which they are handled by the same RTS and running on the same Java Virtual Machine (JVM) instance. One or more subsystems form a SystemJ (GALS) program, in which they may be residing on one computing
Background and Related Works

machine or distributed across many. CDs residing on different SystemJ subsystems communicate through the abstracted channel mechanism and all data exchanged by communicating CDs will be sent and received physically through physical interfaces referred as links (which can be implemented in any types of physical interfaces e.g. TCP/IP, USB, etc.). Figure 2.2 illustrates an example of a SystemJ program which comprises four subsystems (SS1-SS4), with the corresponding subsystems deployed and distributed across three different machines (M1-M3) with links physically connecting pairs of the subsystems.

![SystemJ (GALS) Program](image)

Figure 2.2. Graphical illustration of an example of a SystemJ program

SystemJ has built-in synchronous and asynchronous kernel statements as described in Table 2.1 and Table 2.2, respectively (recreated from [61]).

Table 2.1. SystemJ synchronous kernel statements and their descriptions

<table>
<thead>
<tr>
<th>SystemJ Kernel Statements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>dummy statement</td>
</tr>
<tr>
<td>pause</td>
<td>dummy statement</td>
</tr>
<tr>
<td>[output][input][type] signal S</td>
<td>signal declaration</td>
</tr>
<tr>
<td>emit S (exp)</td>
<td>signal emission</td>
</tr>
<tr>
<td>p1:p2</td>
<td>sequential statement</td>
</tr>
<tr>
<td>while(true){p}</td>
<td>infinite loop</td>
</tr>
<tr>
<td>present (S) {p1} else {p2}</td>
<td>conditional statement</td>
</tr>
<tr>
<td>[weak] abort (immediate) S {p}</td>
<td>preempt watchdog</td>
</tr>
</tbody>
</table>
Table 2.2. SystemJ asynchronous kernel statements and their descriptions

<table>
<thead>
<tr>
<th>SystemJ Kernel Statements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output [type] channel C</td>
<td>output channel port declaration</td>
</tr>
<tr>
<td>input [type] channel C</td>
<td>input channel port declaration</td>
</tr>
<tr>
<td>p1&gt;&lt;p2</td>
<td>asynchronous parallel</td>
</tr>
<tr>
<td>send C ([exp])</td>
<td>sending via channel port C</td>
</tr>
<tr>
<td>receive C ([exp])</td>
<td>receiving via channel port C</td>
</tr>
</tbody>
</table>

2.4.3 DSystemJ

DSystemJ is a system-level language extended from SystemJ to introduce the features to handle dynamic behaviours in distributed systems. DSystemJ introduces the features of dynamic creation and weak migration of CD. While SystemJ has been amended to allow one SystemJ program to be composed from multiple subsystems which can be deployed separately into one or more machines, the feature of subsystem encapsulation doesn’t exist in DSystemJ, and thus, software behaviours in one DSystemJ program can’t be distributed across multiple machines. In addition, compared to SystemJ which has been made compliant with the delayed signal semantics, DSystemJ signal still follows the instantaneous signal semantics like Esterel. However, different from SystemJ channel which enforces only point-to-point communication, DSystemJ channel semantics allow DSystemJ channel to perform one-to-many and many-to-one communications.

2.4.4 LibGALS and LibDGALS

LibGALS is a software programming library which is written entirely in C and extends the C-programming language library particularly for designing Globally Asynchronous Locally Synchronous (GALS) system. Like SystemJ, LibGALS is also based on GALS MoC and adopt similar programming constructs and reactive statements in Esterel and SystemJ to employ similar features of both asynchronous and synchronous concurrency and behaviour reactivity. Apart from the adopted constructs and statements, LibGALS also provides its own programming constructs in the form of Application Programming Interface (API) calls to allow some additional features for creating processes statically and dynamically. Similar to DSystemJ which is extended from
Background and Related Works

SystemJ, LibGALS is also further extended with some additional features for dynamic behaviours, resulting in what is called LibDGALS. Similar to SystemJ and DSystemJ, synchronous behaviours in LibGALS and LibDGALS, also called reactions, advance in lock-step and bound within the same tick-boundary of clock domain. Clock domains have their own notion of ticks and they compose a GALS program. The same communication mechanisms in SystemJ and DSystemJ are also found in LibGALS and LibDGALS. Reactions communicate with the environment and other reactions in the same CD via signal, and they communicate with other reactions in different CDs through channel. Similar with Esterel, signal in LibGALS and LibDGALS also follows the instantaneous signal semantics.

2.5 Combination of Different Software Paradigms

There have been also efforts which involve developing new approaches by synergizing different programming paradigms, utilizing the features available in the original paradigms that the new approaches have advantages compared to the original paradigms. The following sections mention some of the approaches based on the synergy of different paradigms. Among the existing ‘combined’ approaches are the Holonic and MAS, SOA and MAS, SOA and industrial standard approaches, and formal MoC and industrial standard approaches.

2.5.1 Holonic and MAS

The MAS paradigm also enjoy the complementary addition of the holonic paradigm which introduces and promote flexibility, organisation, hierarchy, and reconfigurability of agents. The term ‘holon’ was introduced by Koestler [130] from the Greek language, ‘holos’ which means ‘whole’ and ‘on’ which means ‘part’. The concept of holon is used by Koestler to describe the evolution of biological and social systems. Thus, according to the holonic paradigm, holons are individual ‘fractal’ entities which may consist of multiple ‘holons’ as sub-entities, while at the same time, they are also parts of a greater whole. Some approaches which are based on MAS enhanced with holonic paradigm, for example, ANEMONA [131] and ADACOR [132]. Both ANEMONA and ADACOR use JADE framework to implement their agents.

2.5.2 SOA and MAS

This approach makes use of the features of MAS and SOA paradigm. There are three different methodologies on how the use of both SOA and MAS can be achieved. The first methodology is by creating a proxy function which provides interoperability between SOA and MAS. Some examples which use this paradigm are the WS2JADE [133], AgentWeb Gateway [134], and JADE-JBossESB [135]. The second methodology is by wrapping software agents and exposing
Background and Related Works

them as service entities, for example, the approach of wrapping agents as web services as elaborated in [136]. The third methodology is by creating what is called as service-oriented multi-agent systems (SOMAS) approach. Some examples of SOMAS frameworks are FUSION@ [137] and Thomas [138, 139]. While Thomas specifies the use of FIPA standard in its agent architecture, FUSION@ specification doesn’t follow the FIPA standard. On the other hand, JIAC V combines the first and third methodologies by having dynamic discovery and message exchange mechanism between agents, while also providing gateway function that enables agents to interact with OSGi bundles and communicate with web services.

2.5.3 SOA and Industrial Standard Approaches

The SOA-based approaches have also caught the attention of researchers working in the domain of the industrial standard approaches. While SOA theoretically can be combined with the IEC 61131, the IEC 61131 is not aimed towards creating distributed systems. This is considered as a hindrance in achieving the combination of SOA and IEC 61131 [109, 140]. On the other hand, there have been efforts to combine SOA with the IEC 61499. A conceptual IEC 61499-based PLC programming support environment which adopts the SOA paradigm has been proposed in [141], however the work terminates at the design level and there is not much detail regarding how it will be implemented. The works of [142, 143] attempt to implement SOA using IEC 61499 function blocks, however they do not integrate their SOA implementation into the runtime environment level. The work of [144] integrates IEC 61499 with web services for IEC 61499 function block to make use of the SOA features in web services, however, the approach requires function blocks which are encapsulated as web services and have to be deployed in the cloud, which means that the SOA features used in the approach can only exist in the cloud. Another work [145] attempts to introduce SOA functionality into the IEC 61499 within the runtime level. Despite there have been efforts in synergizing SOA with IEC 61499, it remains to be seen whether they will be adopted and officially included into the IEC 61499 standard.

2.5.4 Formal MoC and Industrial Standard Approaches

The IEC 61131 and IEC 61499 standard are designed with little or no consideration for having any underlying formal semantics and MoC. Currently, the possibility of bringing formal semantics and MoC into both standard has become a domain for research. There have been several efforts in regards to introducing formal semantics and MoC into industrial standard approaches. The work [146, 147] attempt to introduce synchronous semantics into the IEC 61499 by mapping IEC 61499 function blocks into a subset of Esterel. Another work [148] attempts to model distributed IEC 61499 systems using the GALS MoC. Other efforts attempt to introduce other types of formal
Background and Related Works

MoC to IEC 61499, such as using the Net Condition/Event Systems model [149], interacting automata [150], timed automata [151], and Mealy finite state machines [152]. On the other hand, some efforts in introducing formal semantics and MoC into IEC 61131 are also available. For example, the work [153] attempts to introduce synchronous data flow semantics used in SIGNAL language [154] into IEC 61131, [155] attempts to enable mapping of IEC 61131 design written in function block diagram into timed automata, and [156] allows the mapping from timed automata into IEC 61131.

2.6 Qualitative Comparisons of State of the Art

In this chapter, the state of the art design and programming approaches for DISS are qualitatively compared based on the identified design and programming requirements explained in Section 1.3.

2.6.1 Functional Correctness

Typical MAS frameworks such as JADE, JIAC, JADEX, JACK, and Agent Factory do not have underlying formal semantics and MoC, hence they provide limited or nil support for developing correct-by-construction software behaviours. Similarly, SOA approaches also provide little support for developing correct-by-construction software behaviours since they do not possess any underlying formal semantics and MoC. Likewise, the original industrial standard approaches IEC 61131 and IEC 61499 are not based on any formal semantics and MoC, and so are the combined approaches of SOA and MAS, SOA and industrial standard approaches, and Holonic and MAS. Despite the aforementioned attempts in Section 2.5.4 to introduce formal semantics and MoC to the industrial standard approaches, it remains to be seen whether the above works will be included and standardized into the IEC 61131 and IEC 61499 standards.

On the other hand, the aforementioned formal MoC-based approaches Esterel, SystemJ, DSystemJ, and LibGALS and LibDGALS have underlying formal semantics and MoC which facilitate the development of correct-by-construction software behaviours. With underlying formal semantics and MoC, software behaviours in Esterel, SystemJ, DSystemJ, LibGALS, and LibDGALS are formally verifiable.

2.6.2 Dynamicity

Some MAS frameworks such as JADE, JIAC, JADEX, and Agent Factory support handling dynamic behaviours and reconfiguration to certain extent. JADE and JADEX have the programming constructs to allow programmers to dynamically create agents, temporarily stop execution of agents (suspend), resume execution of agents, remove and terminate agents, and also
migrate (strong migration) agents from one execution platform to another execution platform during runtime. JIAC runtime environment provides the support for agent creation, strong and weak migration, and termination of execution of agents without affecting the execution of other existing agents through the adoption of FIPA standard to certain extent. Agent Factory also uses the FIPA standard to allow dynamic creation, suspension, resumption, termination, and weak migration of agents. In regards to behaviour awareness, JADE, JADEX and Agent Factory use the FIPA standard which has the yellow and white page functions. The yellow and white pages contain the list of agents and their descriptions. Meanwhile, unlike JADE, JIAC, JADEX, and Agent Factory, JACK offers limited or nil support and programming constructs to enable programmers to deal with dynamic behaviours and reconfiguration.

While the SOA paradigm itself promotes the concept of loose and dynamic coupling between service entities which facilitates behaviour awareness, and the concept of which service entities (and consequently their services) may join and leave at will during runtime, the SOA paradigm doesn’t specify and define the mechanisms of how the above features can be achieved. These characteristics are also reflected on typical SOA approaches. Typical SOA approaches focus mostly on how to introduce and implement SOA mechanisms with less emphasis on providing any mechanisms on how to perform and handle dynamic behaviours, i.e., to perform dynamic creation, suspension, resumption, termination, and migration of service entities. However, SOA-based approaches like WS4D support handling of dynamic behaviour to certain extent, i.e., creation of termination of service entities.

The IEC 61131 was proposed with little emphasis towards introducing any features for dynamicity due to the fact that conventional industrial manufacturing systems had little or no requirement for dynamic behaviours and reconfiguration [157]. Thus, there is no specification and mechanisms defined in the IEC 61131 standard on how to perform dynamic behaviours and reconfiguration. Despite that previously the industrial manufacturing domain had little or no requirement for dynamic behaviours and reconfiguration, the rise of the flexible [158] and reconfigurable [159, 160] industrial manufacturing domain and the IoT and cloud technology cause a shift in industrial manufacturing requirements, i.e., the requirement to perform dynamic behaviours and reconfiguration on the fly arises. Following this, the successor of the IEC 61131, the IEC 61499, begins to give more attention and orientation towards introducing the mechanisms for dynamic behaviours and reconfiguration into the standard. Currently, the IEC 61499 standard defines a special function block called the management function block which can accept command to perform dynamic behaviour and reconfiguration on function blocks which are connected to the corresponding management function block. The management function block accepts a list of
Background and Related Works

commands including Create, Delete, Start, Stop, Reset, and Kill. The Create and Delete command allow for the creation and removal of function blocks and their interfaces, respectively. The Start and Stop command enables the initiation and temporary halt of function block execution, respectively. The Reset command allows the function block to restart its execution from the beginning, while the Kill command terminates the function block. However, dynamic behaviours and reconfiguration performed by management function block are limited only to function blocks which are connected to the corresponding management function block only. Also, the feature to handle migration of function blocks is not available in the management function block.

Most of the formal MoC-based approaches don’t provide the support for handling dynamic behaviours and reconfiguration to certain extent. For instance, both Esterel and SystemJ don’t have any features or programming constructs which allow programmers to perform dynamic behaviours and reconfiguration. Similar to SystemJ, LibGALS allows the development of static GALS programs and has limited features to perform dynamic behaviours and reconfiguration. On the other hand, as an extended approach from SystemJ, DSystemJ, provides programming constructs that allow performing dynamic behaviours and reconfiguration to certain extent. DSystemJ allows for dynamic creation of clock domains and weak migration of clock domains. Similarly, LibDGALS is an extended version of LibGALS which introduce additional programming constructs to perform certain level of dynamic behaviours and reconfiguration, which includes dynamic creation and termination of CDs. While the work [123] presents a scenario that demonstrates how strong migration can be performed using LibDGALS, the mechanisms for strong migration in the work have to be implemented by the authors themselves and therefore they are responsible in ensuring that the migration data transfer process and all data and execution state are retained to allow proper resumption of the migrating software behaviour’s execution, i.e., clock domain, in the new location. LibDGALS itself doesn’t provide built-in programming constructs and abstracted mechanisms for the programmers to perform either weak or strong migration.

The combined SOA and MAS approaches use the features of SOA in providing behaviour awareness and dynamic coupling between software behaviours in MAS approaches. The holonic paradigm in the combined Holonic and MAS approaches introduces the additional features for self-orchestration and reconfiguration of agents. Similar features of behaviour awareness and dynamic coupling between software behaviours are also introduced by the SOA paradigm in the combined SOA and industrial standard approaches. On the other hand, formal MoC doesn’t introduce additional capability of dynamic behaviours and reconfiguration to the industrial standard approaches.
2.6.3 Concurrency

Typical MAS frameworks, including JADE, JIAC, JADEX, JACK, and Agent Factory, support asynchronous concurrency only which exists between individual agents. Similar nature is also present in SOA approaches such as the OSGi, UPnP, and DPWS-based approaches which asynchrony is natural between individual service entities. Consequently, programmers have to spend more time and efforts to achieve safe synchronization by utilizing access-control mechanisms such as mutexes to implement synchronization in these approaches. This may become increasingly more difficult and complex when more distributed and concurrent software behaviours are involved, potentially leading to a more error-prone implementation [161]. The IEC 61131 isn’t designed to target distributed systems, however, the IEC 61499 has an advantage over the IEC 61131 by being catered to target distributed systems and is capable of handling asynchronous concurrency. On the other hand, the works which introduce synchronous concurrency in IEC 61499 [146, 147] provide synchronous concurrency support which is enforced within and between individual function blocks. The work [148] improve the support for asynchronous concurrency in IEC 61499 by introducing the GALS MoC. Meanwhile, similar to typical MAS frameworks, the combined SOA and MAS, SOA and Industrial Standard, and Holonic and MAS approaches support only asynchronous concurrency.

In regards to the formal MoC-based approaches, Esterel supports synchronous concurrency and can’t support asynchronous concurrency. On the other hand, SystemJ is based on the GALS MoC which enables the support for both asynchronous and synchronous concurrency. Similarly, both asynchronous and synchronous concurrency are also supported by DSystemJ, LibGALS, and LibDGALS.

2.6.4 Composability

Some MAS frameworks such as Agent Factory offer single-level composition only. JIAC allows two-level composition. JACK, JADE, and JADEX allow hierarchical composition. The SOA approaches allow for the creation of composite services which can be composed from multiple service entities with each offering different services. Both the IEC 61131 and IEC 61499 also enable only single-level composition. For example, an IEC 61131 program written in FBD can consist of multiple function blocks, and similarly, an IEC 61499 program can comprise multiple function blocks. The combination of SOA and MAS and SOA and industrial standard approaches also provide single-level of composition. The combination of holonic paradigm and MAS allows software agents to have similar characteristics with holons. The combined formal MoC and industrial standard approaches in [146, 147] have the same level of composition as
defined by the IEC 61499 standard, however the approach of [148] that introduces GALS MoC into the IEC 61499 allows the creation of a GALS system which can be composed from multiple asynchronous IEC 61499 programs and individual IEC 61499 programs may contain multiple function blocks which are made compliant to the synchronous model.

In regards to the considered formal MoC-based approaches, a single Esterel can be composed from multiple hierarchical synchronous reactions. However, the GALS MoC-based SystemJ, DSystemJ, LibGALS, and LibDGALS have additional level of composition apart from the hierarchical synchronous reactions in a single CD. DSystemJ, LibGALS, and LibDGALS programs can be composed from multiple asynchronous CDs, in which individual CDs can be composed from multiple hierarchical reactions. Similarly, a single SystemJ CD can also be composed from multiple hierarchical synchronous reactions. However, an additional level of composition exists in SystemJ compared to DSystemJ, LibGALS, and LibDGALS. In SystemJ, a SystemJ subsystem can have one or more CDs and a SystemJ program can consist of one or more subsystems which can be deployed on one or distributed across multiple computing machines.

2.6.5 Abstraction

Some MAS frameworks such as JADE, JIAC, JADEX, JACK, and Agent Factory are based on Java which allows their agents to execute on a wide range of Java-capable execution platforms. However, MAS frameworks typically require additional middleware layer such as [162], Sixth [163], and CORBA [84] to deploy their agents on different execution platforms.

Some SOA approaches like OSGi are based on Java which makes them compatible with different types of Java-enabled execution platforms. The earliest implementation of UPnP is compatible with Microsoft Windows operating systems (OS) only, however, now there are UPnP-based libraries such as [164-166] which allow the deployment of UPnP on different range of platforms running different types of OS. With the SOA programming approach mostly focuses on implementing SOA paradigm, additional middleware layer may be required to handle low-level implementation details such as physical I/O and communication interfaces.

In terms of the formal MoC-based approaches, Esterel programs are typically compiled into C/C++. Since C/C++ compilers are platform-specific, Esterel programs may face the issue of code portability that they have to be compiled with different compilers to be deployed on different execution platforms. Esterel doesn’t provide the mechanism for automatic memory allocation management like Java, hence the programmers are responsible for managing memory allocation during runtime. Memory allocation management can be a huge challenge especially when complex DISS such as the motivating example is considered. Similar issues also exist in LibGALS and
Background and Related Works

LibDGALS which are based on C/C++. Due to the use of C/C++, programmers are forced to use pointers to access data stored in the memory. The use of pointers can be a nightmare for programmers especially when they are dealing with a complex DISS such as the motivating example.

In contrast, SystemJ and DSystemJ use the accompanied SystemJ and DSystemJ runtime support to handle physical I/O and communication interfaces without the need of additional middleware. Unlike Esterel, LibGALS, and LibDGALS, both SystemJ and DSystemJ use Java, hence memory allocation are managed automatically by the JVM during runtime, relieving programmer’s burden from having to manage memory allocation by themselves. Furthermore, unlike Esterel, LibGALS and LibDGALS which are more constrained in terms of their code portability due to the use of C/C++, the use of Java in SystemJ and DSystemJ enables for higher degree of code portability, allowing for the vision of “write once run everywhere” or “compile once run anywhere” [167] closer to reality. This is a huge advantage, especially for DISS like the motivating example which can use different execution platforms with different physical I/O interfacing and communication capabilities.

The industrial standard approaches are developed to target PLC platforms in particular. The IEC 61131 is the legacy approach which is compatible with many PLC platforms from different vendors. On the other hand, there are IEC 61499 runtime environment implementations which have been developed to allow several PLC platforms to execute IEC 61499 programs. Some examples of IEC 61499-compatible PLC platforms are LoyTec [168], Raspberry Pi-based PLC of Conmeleon [169], and Bachmann [170]. However, overall there are less IEC 61499-compliant PLC platforms than IEC 61131-compliant PLC platforms. Compared to the MAS, SOA, and formal MoC-based approaches, the industrial standard approaches have lower-level of abstraction.

In general, all of the ‘combined’ approaches have the same nature with the individual software paradigms that form them. The combined SOA and MAS approaches may need additional middleware layer to handle with different physical I/O interfacing. The similar case also applies to the combined SOA and industrial standard approaches and Holonic and MAS approaches. Similar to the industrial standard approaches, the combined formal MoC and industrial standard approaches are aimed to target PLC platforms.

2.6.6 Reactivity

While agents are capable to react to the environment, typical MAS frameworks like JADE, JIAC, JADEX, JACK, and Agent Factory do not have built-in reactive programming constructs to allow deterministic software behaviour and safe behaviour pre-emption. This condition also
Background and Related Works

applies to the SOA-based approaches which mostly focus on introducing the SOA paradigm and implementing SOA message exchange mechanisms. In regards to the industrial standard approaches, the IEC 61499 function block separates event and data port interfaces while the IEC 61131 doesn’t have event port interfaces, allowing the IEC 61499 function block to handle and react to events better than the IEC 61131. However, the IEC 61499 doesn’t provide any reactive programming constructs to allow safe and correct behaviour pre-emption.

The combination of SOA and MAS approaches has better level of support for software behaviours to react to the environment compared to the SOA approaches due to the inclusion of the MAS paradigm. However, as mentioned previously, the combined MAS and SOA approaches do not provide reactive programming constructs to allow safe behaviour pre-emption and guaranteed determinism within individual software behaviours. Meanwhile, the holonic and SOA paradigm combination have limited or nil programming constructs to enable deterministic software behaviour and safe behaviour pre-emption.

In regards to the combined formal MoC and Industrial Standard approaches, the works [146, 147] introduce synchronous reactive semantics to the IEC 61499 which allow deterministic behaviour and handling reactivity, however, the feature of safe behaviour pre-emption with explicit scope which is available in the non-graphical, formal MoC-based approaches can’t be used in the graphical-based IEC 61499 approach.

In contrast, the built-in reactive programming constructs in Esterel, SystemJ, DSystemJ, LibGALS and LibDGALS provide the necessary support for handling reactivity. SystemJ, DSystemJ, LibGALS and LibDGALS adopt reactive programming constructs like Esterel to handle reactivity, allowing for safe behaviour pre-emption with explicit scope while at the same time guaranteeing deterministic behaviour, i.e., the same sequences of input events will always lead into the same sequences of output actions, irrespective of the order of events/sensed data.

2.6.7 Data-Driven Computation Capability

Some MAS frameworks such as JADE, JIAC, JADEX, JACK, and Agent Factory use Java and therefore have object-orientation capability to perform complex data-driven computation. The capability of data-driven computation for the SOA approaches will depend on how service entities are implemented since the SOA paradigm itself doesn’t specify how service entities are implemented. Both the industrial standard approaches IEC 61131 and IEC 61499 have object-orientation and are capable of performing data-driven computation. While SystemJ and DSystemJ are also capable of performing rather complex data-driven computation using Java, Esterel is most suited for control-dominated systems and doesn’t have the similar capability as SystemJ and
Background and Related Works

DSystemJ to perform complex data-driven computation. LibGALS and LibDGALS also have data-driven computation capability as offered by the C/C++ language, however, as mentioned in Section 2.6.5, the use of pointers may become a nightmare for programmers. The combination between different paradigms will have the same data-driven computation capability possessed by the individual programming paradigms that form them.

2.7 Summary and Conclusions

This chapter identifies a number of software design and programming paradigms, which include the MAS, SOA, industrial standard approaches, formal MoC-based approaches, and combination of different software paradigms, where the combination of different software paradigms synergize individual software paradigms of MAS, SOA, industrial standard approaches, and formal MoC-based approaches. In this chapter, five MAS frameworks, (JADE, JIAC, JADEX, JACK, and Agent Factory), three SOA approaches (OSGI, UPnP, and DPWS), two industrial standard approaches (IEC 61131, IEC 61499), five formal MoC-based approaches (Esterel, SystemJ, DSystemJ, LibGALS, and LibDGALS), and four different combination of software paradigms (SOA and MAS, Holonic and MAS, SOA and industrial standard approaches, and formal MoC and industrial standard approaches) are evaluated and then compared based on the identified design and programming requirements explained in Section 1.3. The comparison based on the identified design and programming requirements is described in Section 2.6 and summarized in Table 2.3 and Table 2.4.

From the analysis and comparison between the state of the art of design and programming approaches for DISS, it is clear that none of the considered state of the art of design and programming approaches is able to satisfy all of the identified design and programming requirements. The formal MoC-based approaches such as Esterel, SystemJ, DSystemJ, LibGALS, and LibDGALS show good potentials with their underlying formal semantics and MoC. The underlying synchronous semantics and MoC of Esterel is not sufficient for DISS. Though LibGALS and LibDGALS have underlying GALS MoC, the absence of dynamic memory allocation management function and the use of pointers due to the use of C/C++ language may introduce huge burden for programmers when programming complex distributed systems like DISS. While DSystemJ provides programming constructs that allow performing dynamic behaviours to some extent, it is still based on the instantaneous signal semantics and individual software behaviours in one DSystemJ program cannot be distributed across different machines. Meanwhile, despite SystemJ doesn’t have any programming constructs to perform dynamic behaviours, it uses the delayed signal semantics and allows individual software behaviours to be
Background and Related Works

distributed across different computing machines. Except dynamicity, SystemJ is able to satisfy the rest of the design and programming requirements of DISS.

On the other hand, SOA is a software programming paradigm which can be adopted into other software paradigm, similar to how SOA is being adopted in the combined SOA and MAS, and SOA and industrial standard approaches. Since SystemJ lacks the mechanisms to handle dynamic behaviours, the SOA paradigm can be combined with SystemJ to introduce behaviour awareness into SystemJ as an initial step to enable handling dynamic behaviours. The combination of SOA paradigm and SystemJ is expected to facilitate dynamic orchestration of individual software behaviours. Following this observation, Chapter 3 of this thesis proposes a novel design and programming approach which is formed through the synergy of the SOA paradigm and the system-level language SystemJ.
Table 2.3. Comparison between existing design and programming approaches in terms of the identified design and programming requirements (except dynamicity)

<table>
<thead>
<tr>
<th>Programming Paradigm</th>
<th>Programming Framework</th>
<th>Design and Programming Requirements</th>
<th>Functional Correctness</th>
<th>Concurrency</th>
<th>Composability</th>
<th>Abstraction</th>
<th>Reactivity</th>
<th>Data-Driven Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAS</td>
<td>JADE</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Hierarchical</td>
<td>Java-enabled, additional middleware may be necessary</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>JIAC</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>Java-enabled, additional middleware may be necessary</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JADEX</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Hierarchical</td>
<td>Java-enabled, additional middleware may be necessary</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JACK</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Hierarchical</td>
<td>Java-enabled, additional middleware</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>-------------------------</td>
<td>-----------------------------------</td>
<td>------------------------------</td>
<td>----------------</td>
<td>----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agent Factory</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>Java-enabled, additional</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>middleware may be necessary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOA</td>
<td>OSGI</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>Java-enabled,</td>
<td>Limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>additional middleware</td>
<td>additional middleware</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>may be necessary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPnP</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>Java-enabled, additional</td>
<td>Limited</td>
<td>Implementation-depant</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>middleware may be necessary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPWS</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Two-level</td>
<td>Java-capable, additional</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>middleware may be necessary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Industrial Standard Approaches</td>
<td>IEC 61131</td>
<td>Limited</td>
<td>Limited</td>
<td>Single-level</td>
<td>IEC 61131 Runtime Environment, Lower programming abstraction</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------</td>
<td>---------</td>
<td>---------</td>
<td>--------------</td>
<td>----------------------------------------------------------</td>
<td>---------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>IEC 61499</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>IEC 61499 Runtime Environment, Lower programming abstraction</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formal MoC-based Approaches</td>
<td>Esterel</td>
<td>Supported by SR MoC (*)</td>
<td>Synchronous only</td>
<td>Hierarchical</td>
<td>Limited code portability, no dynamic memory allocation management</td>
<td>Supported</td>
<td>Limited</td>
<td></td>
</tr>
<tr>
<td>SystemJ</td>
<td>Supported by GALS MoC</td>
<td>Asynchronous and Synchronous</td>
<td>Hierarchical</td>
<td>Java-enabled, no additional middleware needed</td>
<td>Supported</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>Supported by</td>
<td>Asynchronous and Synchronous</td>
<td>Hierarchical</td>
<td>Java-enabled, no additional middleware needed</td>
<td>Supported</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>-----------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------</td>
<td>-----------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSystemJ</td>
<td>GALS MoC (*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LibGALS</td>
<td>GALS MoC (*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LibDGALS</td>
<td>GALS MoC (*)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUSION@</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Agent</td>
<td>Agent framework-dependant</td>
<td>Limited</td>
<td>Agent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOA and MAS Approach</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Agent</td>
<td>Agent framework-dependant</td>
<td>Limited</td>
<td>Agent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thomas</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Agent</td>
<td>Java enabled, need additional middleware</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JIAC V</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Multi-level</td>
<td>Java enabled, need additional middleware</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------</td>
<td>---------</td>
<td>-------------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>---------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>Holonic and MAS</td>
<td>ADACOR</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Hierarchical</td>
<td>Java-enabled, need additional middleware</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANEMONA</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Hierarchical</td>
<td>Java-enabled, need additional middleware</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SOA and Industrial Standard Approaches</td>
<td>SOA + IEC 61499</td>
<td>Limited</td>
<td>Asynchronous only</td>
<td>Single-level</td>
<td>Same with IEC 61499</td>
<td>Limited</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Formal MoC and Industrial Standard Approaches</td>
<td>Synchronous MoC + IEC 61499</td>
<td>Supported by SR MoC</td>
<td>Synchronous and Asynchronous</td>
<td>Single-level</td>
<td>Same with IEC 61499</td>
<td>Supported</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GALS MoC + IEC 61499</td>
<td>Supported by GALS MoC</td>
<td>Synchronous and Asynchronous</td>
<td>Single-level</td>
<td>Same with IEC 61499</td>
<td>Supported</td>
<td>Supported</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.4. Comparison between existing design and programming approaches in terms of the dynamicity requirement

<table>
<thead>
<tr>
<th>Programming Paradigm</th>
<th>Programming Framework</th>
<th>Design and Programming Requirement : Dynamicity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Behaviour Creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strong Migration</td>
</tr>
<tr>
<td>MAS</td>
<td>JADE</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>JIAC</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>JADEX</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>JACK</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>Agent Factory</td>
<td>Supported</td>
</tr>
<tr>
<td>SOA</td>
<td>OSGI</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>UPnP</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>DPWS</td>
<td>Supported</td>
</tr>
<tr>
<td>Industrial Standard Approaches</td>
<td>IEC 61131</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>IEC 61499</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Esterel</td>
<td>Limited</td>
</tr>
<tr>
<td>Formal MoC-based Approaches</td>
<td>SystemJ</td>
<td>Limited</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>DSystemJ</td>
<td>Supported</td>
<td>Limited</td>
</tr>
<tr>
<td>LibGALS</td>
<td>Limited</td>
<td>Limited</td>
</tr>
<tr>
<td>LibDGALS</td>
<td>Supported</td>
<td>Limited</td>
</tr>
<tr>
<td>SOA + MAS</td>
<td>FUSION@</td>
<td>Agent framework-dependant</td>
</tr>
<tr>
<td>Thomas</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>JIAC V</td>
<td>Supported</td>
<td>Limited</td>
</tr>
<tr>
<td>Holonic + MAS</td>
<td>ADACOR</td>
<td>Supported</td>
</tr>
<tr>
<td>ANEMONA</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>SOA and Industrial Standard Approaches</td>
<td>SOA + IEC 61499</td>
<td>Supported</td>
</tr>
<tr>
<td>Formal MoC and Industrial Standard Approaches</td>
<td>Synchronous MoC + IEC 61499</td>
<td>Supported</td>
</tr>
<tr>
<td>GALS MoC + IEC 61499</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Synchronous data flow MoC + IEC 61131</td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>
3 SOSJ : The Integration of SOA Paradigm with SystemJ

As explained in Section 2.7, SystemJ possesses features which are able to satisfy most of the design and programming requirements of DISS. However, SystemJ is designed to target static distributed systems and doesn’t possess programming constructs and mechanisms to deal with dynamic behaviours. On the other hand, SOA promotes the concept of dynamic, loose coupling of software behaviours which facilitates dynamic orchestration between software behaviours. Based on this observation, a novel programming paradigm called Service Oriented SystemJ (SOSJ) is proposed. This programming paradigm is achieved through the synergy of SOA paradigm and the system-level language SystemJ. The original SystemJ Runtime Support is extended to achieve the SOA-based interaction of Publish, Find, and Bind services shown in Figure 2.1. The extension introduces SOA functionalities which allow CD to discover, advertise, and invoke services dynamically.

To realize the synergy of SOA paradigm with SystemJ, a number of new elements and features have to be introduced into SystemJ. First, the concept of service entity in SOA which use or provide services are adopted into SystemJ software behaviour. Second, similar to service entity in SOA, a CD needs to have service description which provides additional information regarding functionalities (services) offered by a CD. Third, the information exchange mechanism as one feature provided by the SOA paradigm needs to be integrated with SystemJ. Fourth, the feature of loose-coupling to enable dynamic orchestration of software behaviours in the SOA paradigm needs to be implemented. This chapter elaborates how the concepts and features in SOA are adopted by SystemJ to realize the above features. This chapter is organized as follows. Section 3.1 considers a new perspective on how to look at software behaviours from the context of both the SOA paradigm and SystemJ. Following this perspective, new provisions are proposed in the following sections. Section 3.2 introduces service description in SOSJ and how it is structured to define and provide information regarding services and their interfaces for service consumer to perform service
invocation. Section 3.2 also introduces service registry, which has a data structure that stores service description and its structure. Next, Section 3.3 explains the underlying SOA functionalities in SOSJ, which consist of Beacon, Advertisement, Request for Advertisement, Discovery, Discovery Reply, and Notify to realize the SOA-based Publish and Find interactions illustrated in Figure 2.1. Finally, Section 3.4 describes the necessary provisions to allow service invocation, therefore realizing the Bind interaction depicted in Figure 2.1.

3.1 Software Behaviours from the perspective of SOA and SystemJ (SOSJ)

As previously mentioned, the SOA paradigm uses the terminology of service entity to refer to software behaviours. On the other hand, in SystemJ, CD is a software behaviour which is composed from one or more synchronous reactions. The synergy of both SOA paradigm and SystemJ leads to adoption of the nature of service entity in SOA (i.e. to use or provide services) into SystemJ.

With this synergy, a reaction may possess service consumer role (which uses services), service provider role (which provides services), or both (due to the possibility of nested reactions). Since CD is composed from one or more synchronous reactions, CD will have the same roles as the synchronous reactions that compose it, and therefore, unlike SOA which sees service entity as software behaviour that either uses or provides services, SystemJ allows CD to have mixed service provider and consumer roles. However, since the synchronous reactive MoC is enforced within individual CDs in SystemJ to preserve determinism and guarantee functional correctness, individual reactions are constrained to the ‘tick boundary’ of CD and thus are considered as integral parts of CD. Therefore, reaction can’t be considered as an exact equivalent of service entity, since service entity in SOA is not considered to have such limitation. Based on the reactions that compose it, there are three different scenarios for a CD:

1) CD is composed from one or more reactions which all have service consumer role only.
2) CD is composed from one or more reactions which all have service provider role only.
3) CD is composed from one or more reactions with a mix of service provider and service consumer roles.

Also, it is possible that some or all synchronous reactions in the CD are connected with other reactions in the same CD (via signals) or other CDs (via channels) to produce new services created by composition of individual services offered by individual reactions. In scenario 3), since individual reactions in a CD can be composed from hierarchical child reactions, reactions in the
same CD may also possess mixed service provider and consumer roles due to their child reactions which may consist of service provider role, service consumer role, or a mix of both. The examples of scenarios 1), 2) and 3), and creation of composite service are illustrated in Figure 3.1. Note that since SystemJ uses the abstracted communication mechanisms of signal and channel, the implementation of service interfaces in SOSJ will adopt the same abstraction.

As shown in Figure 3.1., the CDs illustrated by the examples on the top-left, top-right, and bottom part can be in the same subsystem or distributed in different subsystems. The top-left part can represent scenario 1) or 2). If scenario 1) applies, reactions R11 and R12 have service consumer role. In this case, R11 can invoke services offered by R21 via channel Ch1121 and receives results via channel Ch2111. Alternatively, if scenario 2) applies, R11 and R12 have service provider role, while R21 and R31 have service consumer role. In this case, R21 can invoke the services offered by R11 via channel Ch2111 and receives results via channel Ch1121. Meanwhile, the top-right part illustrates CD4 which forms an example of scenario 3). For instance, reaction R41 and R43 have service provider role, while R42 and R44 have service consumer role. Having service consumer role, R51 can invoke the services offered by R41 via Ch5141 and receive results via Ch4151. Finally, the bottom part illustrates an example of the creation of composite service. For example, reaction R73 acts as an orchestrator that forms a composite service through the composition of itself with R71 via signals. As a service consumer, R91 can invoke the composite service via Ch9173 and receives results via Ch7391.

Figure 3.1. Illustration showing examples of scenario 1), 2), 3), and composite service.
With the adoption of SOA paradigm, the bindings of channels used by reactions in different CDs are made to be reconfigurable to allow dynamic interactions/aggregations of reactions in different CDs. This feature is also needed when any change of the presence of CD occurs. It is worth noting that despite the aforementioned changes in the nature of CDs due to the SOA paradigm, the nature of subsystem remains unchanged (i.e. subsystem encapsulates one or more CDs handled by the same RTS and JVM). The nature of signal is also unchanged since signal is a broadcast-like mechanism which, once emitted, will be visible by all reactions in the same CD (no need for reconfiguration of signals).

By considering the interactions shown in Figure 2.1 and also the nature of reaction and CD, it can be inferred that services offered by reactions with service provider role have to be advertised to other reactions that have service consumer role to be able to dynamically find and invoke services during runtime. However, this can’t be achieved without service description which provides the description of individual services and also the service registry (which stores the service description of all services).

### 3.2 Service Description and Service Registry

This chapter introduces and elaborates service description to provide the description of individual services to be advertised and found by service consumer in SOSJ. Service description may contain information such as the capability and specification of services offered by service providers, service interfaces which are used to access and use the provided services, and the location of the service providers.

The existing SOA approaches such as UPnP and DPWS describe their service description using mainstream markup languages such as XML and WSDL. However, it should be noted that the SOA paradigm itself doesn’t specify any particular standards on how service description should be implemented. In case of SOSJ, programmers write service description in XML format, the same format already used to describe SystemJ CD (including signal and channel) configuration [58]. This allows the reuse of existing XML parser functionality in the SystemJ RTS which is used to parse CD and configuration (it will be elaborated in Section 4.2). Since service description may contain different types of information, service description in SOSJ written by programmers consists of a number of XML elements and attributes.

To show how service description in SOSJ is structured, Listing 3.1 shows an example of service description in SOSJ which needs to be written by programmers in XML format. The example depicts a service description which describes services offered by reaction(s) in a CD.
SOSJ: The Integration of SOA Paradigm with SystemJ

together with SystemJ

The CD governs the behaviour of conveyor CB1 in the motivating example shown in Figure 1.1. Note that service description is typically application-dependent and also depends on the nature of the services themselves.

Listing 3.1. Example of service description.

```xml
<Services>
  <serviceDescription serviceName="CB1Service" associatedCDName="CB1CD">
    <physicalParameters>
      <parameter name="length" unit="centimeter" value="400" />
    </physicalParameters>
    <action>
      <actionAttributes name="ActuateForward" description="actuate conveyor CB1 forward">
        <actionParameters>
          <parameter name="speed" unit="rev/s" minValue="1" maxValue="40" />
        </actionParameters>
        <actionInterfaces>
          <interface name="CB1ReqCh1" type="channel" direction="input" />
          <interface name="CB1ReqS1" type="signal" direction="input" />
          <interface name="CB1RespCh1" type="channel" direction="output" />
          <interface name="CB1RespS1" type="signal" direction="output" />
        </actionInterfaces>
      </actionAttributes>
    </action>
    <action>
      <actionAttributes name="ActuateBackward" description="actuate conveyor CB1 backward">
        <actionParameters>
          <parameter name="speed" unit="rev/s" minValue="1" maxValue="40" />
        </actionParameters>
        <actionInterfaces>
          <interface name="CB1ReqCh2" type="channel" direction="input" />
          <interface name="CB1ReqS2" type="signal" direction="input" />
          <interface name="CB1RespCh2" type="channel" direction="output" />
          <interface name="CB1RespS2" type="signal" direction="output" />
        </actionInterfaces>
      </actionAttributes>
    </action>
  </serviceDescription>
</Services>
```

Referring to Listing 3.1, there are a number of attributes in the service description. The **serviceName** attribute (line 2) defines the identifier of service description that describes services offered by reactions in the corresponding CD. The **associatedCDName** attribute (line 2) defines the name of CD which the service description is associated to. Each service description of a particular serviceName has the following elements and attributes. The **physicalParameters** element contains some description of physical specification of e.g. sensors, actuators, machines, whose behaviours are governed by the reactions grouped in the CD. In the example, the **physicalParameters** element describes a specification that states the length of the conveyor CB1 (line 4). The **action** attribute contains the description of the services offered by reactions
encapsulated and grouped in a CD name defined in the associatedCDName attribute. Note that there can be multiple action elements which reflect the number of services offered by the reaction(s) in the CD. The name attribute (line 7, line 20) defines the identifier of the action. The description attribute (line 7, line 20) is used to give more context regarding the purpose of the corresponding service.

The example shows that the reactions in CB1CD provide two services, named ‘ActuateForward’ and ‘ActuateBackward’. The actionParameters element defines the capability of the service. The example states that both the ‘ActuateForward’ and ‘ActuateBackward’ have one parameter named ‘speed’ with unit ‘rev/min’ (revolution per minute) ranging from 1-40 rev/min, specifying the rotational speed of conveyor the service is able to perform (line 9, line 22). The actionInterfaces element defines the interface in the CD used for invoking the provided services. In the example, the ‘ActuateForward’ action has four interfaces, consisting of a pair of input and output channel ports (CB1ReqCh1 and CB1RespCh1) and also a pair of input and output signals (CB1ReqS1 and CB1RespS1), as shown in line 12-15. Similarly, the ‘ActuateBackward’ action also has four interfaces, consisting of a pair of input and output channel ports (CB1ReqCh2 and CB1RespCh2) and also a pair of input and output signals (CB1ReqS2 and CB1RespS2) (line 25-28). Input signals and input channels defined within the actionInterfaces element are used to accept request to invoke services from service consumer, while output signals and channels are used to send response (which can contain results, if necessary, depending on the nature of services) back to the service consumer.

Writing service description in XML is performed by programmers only during design time. Meanwhile, the content of service description will be stored in JSON format to allow easy manipulation and management of the service description data during runtime. The result of the mapping of service description example in Listing 3.1 is shown in Listing 3.2. Looking at both Listing 3.1 and Listing 3.2, the value of the serviceName attribute in line 2 of Listing 3.1 (“CB1Service”) is mapped to line 1 of Listing 3.2. The associatedCDName attribute in line 2 of Listing 3.1 is mapped to line 2 of Listing 3.2. The elements within the physicalParameters element (line 4, Listing 3.1) are mapped to line 4-7 of Listing 3.2. The name attributes in line 7 and 20 of Listing 3.1 are mapped to line 12 and 46 of Listing 3.2. The description attributes in line 7 and 20 of Listing 3.1 are mapped to line 13 and 47 of Listing 3.2. The elements within the actionParameters element in line 9 and 22 of Listing 3.1 are mapped to line 15-19 and line 49-53 of Listing 3.2, respectively. The elements within the actionInterfaces attributes in line 12-15 and line 25-28 are mapped to line 23-42 and line 57-76 of Listing 3.2, respectively.
SOSJ : The Integration of SOA Paradigm with SystemJ
Services are advertised to service registry, which has a data structure that stores the service
description of the advertised services. Initially, SOSJ is designed for each subsystem to have
service registry which contains the service description of services advertised by all SOSJ CDs3
belonging to all subsystems of a SOSJ program (or global service registry) in the RTS of each
subsystem [171]. However, since it is becoming more challenging to synchronize the content of
among service registries as the number of subsystems increases, the later design of SOSJ moves
towards having a centralized global service registry which is separated from RTS. Instead, each
subsystem has ‘local service registry’ for them to store the description of services offered by CDs
belonging in the corresponding subsystem and also services of interest offered by CDs belonging
in other subsystems. Full details regarding the architecture of SOSJ framework and how SOSJ
RTS is implemented will be explained in Chapter 5 and 6.
Listing 3.2. Mapping result of service description in JSON format.
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

{"CB1Service": {
"associatedCDName": "CB1CD",
"physicalParameters": {
"parameter1": {
"name" : “length”,
"unit" : "centimeter",
“value” : “400”
}
},
“action”: {
“action1”: {
“name” : “ActuateForward”,
“description” : “actuate conveyor CB1 forward”,
“actionParameters” : {
“parameter1” : {
“name”
: “speed”,
“unit”
: “rev/s”,
“minValue” : “0”,
“maxValue” : “40”
},
},
“actionInterfaces” : {
“interface1” : {
“name” : ”CB1ReqCh1” ,
“type” : “channel”,
“direction” : “input”,
},
“interface2” : {
“name” : “CB1ReqS1”,
“type” : “signal”,
“direction” : “input”,
},
“interface3” : {
“name” : ”CB1RespCh1” ,
3

In the context of providing services, since services are offered by reactions which are grouped into CDs, the
expression of ‘services offered by CDs’ will also be used apart from ‘services offered by reactions’.

53


In both global and local service registry, the service descriptions are grouped based on the subsystem where the CDs providing the advertised services belong to. To illustrate how service descriptions are structured in both global and local service registry, Listing 3.3 shows an example of service descriptions which describe services associated to conveyor CB1 (‘CB1Service’), conveyor CB2 (‘CB2Service’), conveyor CB3 (‘CB3Service’), and also conveyor CB4
SOSJ : The Integration of SOA Paradigm with SystemJ

(‘CB4Service’), diverter D1 (‘D1Service’), and photo eye sensor PE1 (‘PE1Service’). The CDs which provide services associated to conveyor CB1, CB2, and CB3 belong to a subsystem named SS1, while the CDs which provide services associated to diverter D1, conveyor CB4, and photo eye sensor PE1 belonging to another subsystem named SS2. As shown in Listing 3.3, the CB1Service, CB2Service, and CB3Service are grouped to SS1 key (line 1-11), while the D1Service, CB4Service, and PE1Service are grouped to SS2 (line 12-22).

Listing 3.3. Example how service description is structured in service registry.

```
1   {"SS1": {"CB1Service": {
2       .... <!-- service description CB1Service -->
3       },
4       "CB2Service": {
5       .... <!-- service description CB2Service -->
6       },
7       "CB3Service": {
8       .... <!-- service description CB3Service -->
9       },
10      ....
11   }
12   "SS2": {"D1Service": {
13      .... <!-- service description D1Service -->
14      },
15      "CB4Service": {
16      .... <!-- service description CB4Service -->
17      },
18      "PE1Service": {
19      .... <!-- service description PE1Service -->
20      },
21      ....
22   }
23   ....
24   }
```

With the introduction of the dynamic behaviour handling functions (which will be described in Chapter 6), the global service registry stores CD macro-states (explanation regarding CD macro-states are described in Chapter 4) and physical address of subsystems in separate new data structures. To show how both the CD macro-state and physical address of subsystems data structures are structured, Listing 3.4 shows an example of CD macro-state data structure in global service registry, while Listing 3.5 shows an example of subsystem’s physical location data structure in the global service registry.

Listing 3.4. CD macro-states data structure in the global service registry.

```
1   {
2       "SS2": {
3           "CB4CD": "Running",
4           "PE1CD": "Running",
```

55
SOSJ: The Integration of SOA Paradigm with SystemJ

Once Advertisement message is received by the global service registry, the information about CD macro-states and physical location of subsystem carried by the Advertisement message are fetched by the global service registry, transformed, and stored in two separate data structures as shown in Listing 3.4 and Listing 3.5, respectively. The CD name and its macro-state are grouped according to the subsystems they belong to, while the physical location of subsystems are listed according to their corresponding subsystem names.

3.3 SOA Functionalities

There are new SOA functionalities which are incorporated to introduce the SOA-based interactions shown in Figure 2.1. Referring to Figure 2.1, Find represents the interaction between service registry and service consumer, Publish represents the interaction between service provider and service registry, and Bind represents the interaction between service consumer and service provider (or also known as service invocation). In SOSJ, Find is enabled by functionalities named Beacon, Discovery, and Discovery Reply, Publish is realized by Beacon, Advertisement, and Request for Advertisement, and Bind corresponds to service invocation performed via signal or channel. The conceptual sequence diagram which describes the aforementioned functionalities used to realize the Publish and Find interactions (except Bind or service invocation, which will be elaborated in Section 3.4) is shown in Figure 3.2.

The following elaborates each SOA functionality in SOSJ, comprising Beacon, Advertisement, Request for Advertisement, Notify, Discovery, and Discovery Reply, from the conceptual point of view and how their messages are structured. These SOA functionalities are inspired from the Simple Service Discovery Protocol (SSDP) [100]. Note that all of the
The Integration of SOA Paradigm with SystemJ

The aforementioned SOA messages are structured in JSON format for relatively straightforward construction of and extraction of information from messages.

Figure 3.2. Conceptual sequence diagram of the proposed SOA protocols which realize Publish and Find.

3.3.1 Beacon

Beacon is a SOA functionality in SOSJ which is used by service registry to inform service provider and service consumer regarding its presence. Referring to Figure 3.2, Beacon message is broadcasted from global service registry upon its initiation and is retransmitted periodically. Periodic transmission of the Beacon message is necessary in order for service provider and consumer which may join at will during runtime to be informed regarding service registry’s presence. The Beacon message contains several information regarding the physical location (e.g. IP Address, URI) of the global service registry. To show the structure of the Beacon message, Listing 3.6 presents an example of a Beacon message, formatted in JSON. As shown in Listing 3.6, Beacon message contains four JSON keys. The ‘msgType’ key (line 2) defines the type of the message being transmitted, which is ‘Beacon’ referring to the Beacon message. The ‘regID’ key (line 3) defines the global service registry identifier, which in the example the global service
registry transmitting the Beacon message has ‘registry1’ as its identifier. The ‘regAddr’ key (line 4) defines the physical location of the global service registry (in this case it’s an IPv4 address of 192.168.1.2) that sends the Beacon message. The ‘retransmissionTime’ key (line 5) notifies the receiving parties regarding the transmission period of the Beacon message, measured in milliseconds. This means that the Beacon message is retransmitted after 20000 milliseconds\(^4\) has elapsed in the global service registry’s side.

Listing 3.6. Example of Beacon message showing the structure of Beacon message.

```
1   {
2     "msgType" : "Beacon",
3     "regID" : "registry1",
4     "regAddr" : "192.168.1.2",
5     "retransmissionTime" : "20000"
6   }
```

### 3.3.2 Advertisement

Advertisement is a SOA functionality used by service provider to register their services to the global service registry. Similar to Beacon, Advertisement is also periodic in nature. Referring to Figure 3.2, initial Advertisement is transmitted from service provider once a Beacon has been received in the service provider side. An Advertisement is valid for a certain period of time (or referred as ‘expiry time’), hence it has to be refreshed before the expiry time elapses. If an advertisement is not refreshed until the expiry time elapses (or until the advertisement is considered ‘expired’), services associated with the advertisement is considered unavailable and therefore their service descriptions are removed from the global service registry (indicated by the ‘RemoveAdv()’ operation in Figure 3.2). Advertisement is also transmitted by service provider as a response after receiving a Request for Advertisement (which will be elaborated in Section 3.3.3). Advertisement message contains the service descriptions of services associated to a particular subsystem, the subsystem’s name, and the physical location (address) of the subsystem. To show the structure of Advertisement message, Listing 3.7 shows an example of an Advertisement message, formatted in JSON.

Listing 3.7. Example of Advertisement message showing the structure of Advertisement message.

```
1   {

4 The time value put in the ‘retransmissionTime’ key is indicative and an approximation to certain extent. This would depend on several factors including the underlying computing platforms, operating systems, and the JVMs where SOSJ is deployed. The aspect of real timeliness of the SOA functionality in SOSJ and whether SOSJ can actually guarantee and achieve the exact periodic time as mentioned in the “retransmissionTime” key is out of the scope of this thesis.
Referring to Listing 3.7, the Advertisement message contains eight JSON keys. The ‘msgType’ key (line 2) defines the type of the message being transmitted, which is ‘Advertisement’ referring to the Advertisement message. The ‘associatedSS’ key (line 3) defines the name of the subsystem where the advertised services are associated to. The ‘SSAddr’ key (line 4) defines the physical location of the subsystem where the CDs offering the advertised services belong to (in this example the physical location of the subsystem is an IPv4 address of 192.168.1.3). The ‘expiryTime’ key (line 5) defines the maximum time length which an advertisement is valid for (expiry time). The ‘notify’ key (line 6) is used by the service provider side to inform the global service registry that there are new services being advertised or existing services have been removed from the corresponding subsystem. The ‘notify’ key may have either of these values, true or false. If the value is true, upon receiving the Advertisement message, the global service registry will transmit a Notify message to service consumer. Otherwise if the value is false, the Advertisement message will be considered by the global service registry as a regular periodic advertisement sent by service provider to refresh their advertisement. The ‘serviceList’ key (line 7) contains the service descriptions of the advertised services. The service description included in the Advertisement message shown in Listing 3.7 refers to the services associated to

---

5 Services are associated to the CDs which offer them, therefore they are also grouped according to the subsystems which the CDs belong to.

6 Similar to the “retransmissionTime” key in the Beacon message, the time value put in the “expiryTime” key is indicative and an approximation to certain extent. Whether SOSJ can actually guarantee and achieve the exact periodic time as mentioned in the “expiryTime” key is out of the scope of this thesis.
conveyors CB1 and CB2 which are shown in line 1-6 of Listing 3.3. The last two keys, changedCDStat (line 16) and CDStat (line 17), are used to inform other parties regarding the CD’s macro-state (which will be explained in Section 4.3) and the name of the subsystem the CDs belong to.

The ‘changedCDStat’ key informs the global service registry whether there are changes in CD macro-state (which reflects whether dynamic behaviour is performed). The value of the changedCDStat is set to ‘true’ if a change in the CD macro-state exists, or false if there is no change in the CD macro-state. Meanwhile, the CDStat contains the macro-states of individual CDs that belong to the corresponding subsystem. Advertisement is transmitted to the global service registry when there is a change in CD macro-state. Note that the CD macro-state information follows the nature of Advertisement, i.e., if an Advertisement isn’t refreshed beyond the expiry time, the CDs associated with the CD macro-state information included in the Advertisement will be considered as ‘unavailable’. When a dynamic behaviour has been performed, Advertisement is re-sent to the global service registry containing updated CD macro-states. For example, when a CD has moved from the Running state to the Suspended state, the CD macro-state information is still included inside the Advertisement message, as shown in line 20, however the service description of services associated with the suspended CD, e.g. CB3CD (which is in the Suspended state, belonging to subsystem SS1), is not included in the Advertisement message.

3.3.3 Request for Advertisement

Referring to Figure 3.2, Advertisement may be lost during transmission and doesn’t reach its intended destination, i.e., global service registry. This may cause the corresponding advertisement to be left unrefreshed until the advertisement expires. Consequently, this will cause the global service registry to assume the services associated with the unrefreshed advertisement to be “unavailable” (after the advertisement expires), despite that the service provider may still exist. Thus, Request for Advertisement is introduced to address this issue. Request for Advertisement is a SOA functionality used by global service registry to contact a particular service provider when its advertisement is approaching expiry to request the corresponding advertisement refreshed (by retransmitting Advertisement) by the service provider. When the Request for Advertisement message is received by the service provider, Advertisement is transmitted from the service provider to the global service registry as a reply to the Request for Advertisement. To show the structure of Request for Advertisement message, Listing 3.8 shows an example of Request For Advertisement message.
SOSJ : The Integration of SOA Paradigm with SystemJ

Listing 3.8. Example of Request for Advertisement message.

```
1 { 
2   "msgType" : "RequestForAdvertisement", 
3   "regID" : "registry1", 
4   "regAddr" : "192.168.1.2", 
5   "destSS" : "SS1",  
6   "respPort" : 200'  
7 } 
```

Referring to Listing 3.8, the Request for Advertisement message contains four JSON keys. The ‘msgType’ key (line 2) defines the type of message being transmitted, which is ‘RequestForAdvertisement’ referring to the Request for Advertisement. The ‘regID’ key (line 3) defines the identifier of the global service registry that sends the Request for Advertisement. The ‘regAddr’ key (line 4) defines the physical address of the global service registry (in this case it’s an IPv4 address of 192.168.1.2). The ‘destSS’ key (line 5) defines the name of the subsystem associated to the Advertisement being requested. Finally, the “respPort” key (line 6) defines the physical IPv4 port of where the Advertisement message (as a response to the Request for Advertisement) needs to be transmitted through.

### 3.3.4 Discovery

Discovery is a SOA functionality used by service consumer to obtain the list of advertised services in the global service registry. Referring to Figure 3.2, the global service registry notifies service consumer regarding its presence using Beacon. Once service consumer receives the Beacon sent from the global service registry, service consumer may query the global service registry for advertised services by performing Discovery. After the Discovery message is received, the global service registry responds with Discovery Reply which returns the service description of all advertised services sent to the requesting service consumer. An example of Discovery message is shown in Listing 3.9.

Referring to Listing 3.9, Discovery message contains six JSON keys. The ‘msgType’ key (line 2) defines the type of message being transmitted (Discovery). The ‘associatedSS’ key (line 3) defines the name of the subsystem which service consumer that performs the Discovery is associated to. The ‘regAddr’ key (line 4) defines the physical address of the global service registry (in the example the address is an IPv4 address of 192.168.1.2). The ‘regID’ key (line 5) defines the identifier of the global service registry which is queried by the service consumer sending the Discovery message. The ‘sourceAddress’ key (line 6) defines the physical address of the service consumer performing the Discovery. Finally, The ‘respPort’ key (line 7) defines the physical IPv4
SOSJ: The Integration of SOA Paradigm with SystemJ

port for the global service registry to use for transmitting the Discovery Reply message to the service consumer as a response to the Discovery message.

Listing 3.9. Example of Discovery message.

```json
1 { 
2   "msgType": "Discovery", 
3   "associatedSS": "SS2",  
4   "regAddr": "192.168.1.2",  
5   "regID": "registry1",  
6   "sourceAddress": "192.168.1.4",  
7   "respPort": "444" 
8 }
```

### 3.3.5 Discovery Reply

Discovery Reply is a SOA functionality used by the global service registry as a response to the Discovery sent to the service consumer that transmits the Discovery. The Discovery Reply returns the service descriptions of all advertised services stored in the global service registry to the service consumer which transmits the Discovery. To show how Discovery Reply message is structured, Listing 3.10 shows an example of a Discovery Reply message.

Listing 3.10. Example of Discovery Reply message.

```json
1 { 
2   "msgType": "DiscoveryReply", 
3   "regAddr": "192.168.1.2", 
4   "regID": "registry1", 
5   "destSS": "SS1", 
6   "destAddr": "192.168.1.1", 
7   "serviceList": {  
8     "SS1": {  
9       "CB1Service": {  
10         ..... // service description CB1Service  
11       },  
12       "CB2Service": {  
13         ..... // service description CB2Service  
14       },  
15       .....  
16     },  
17     "SS2": {  
18       "D1Service": {  
19         ..... // service description D1Service  
20       },  
21       "CB4Service": {  
22         ..... // service description CB4Service  
23       },  
24       "PE1Service": {  
25         ..... // service description PE1Service  
26       },  
27       .....  
28   }
29 }
```
Referring to Listing 3.10, Discovery Reply message contains eight JSON keys. The 'msgType' key (line 2) defines the type of the message, in this case it is ‘DiscoveryReply’ (referring to Discovery Reply). The ‘regAddr’ key (line 3) defines the physical address where the global service registry is located (in the example the address is an IPv4 address of 192.168.1.2). The ‘regID’ key (line 4) defines the identifier of the global service registry that sends the Discovery Reply message. The ‘destSS’ key (line 5) defines the subsystem name of the consumer that performs Discovery. The ‘destAddr’ key (line 6) defines the physical location (address) of the subsystem defined in the ‘destSS’ key. The ‘serviceList’ key (line 7) contains the service descriptions of services which are advertised and stored in the global service registry. The example of service description included in the ‘serviceList’ key in Listing 3.10 has the same content with the example of service description shown in Listing 3.3. The CDStat (line 29) and SSAddr (line 41). The CDStat key includes the macro-state of individual CDs that belong to different subsystems. The SSAddr key contains the information of the physical location (address) of individual subsystems. Note that the example of Discovery Reply message in Listing 3.10 is correlated with the Advertisement message example shown in Listing 3.7 by having the CB3CD in the Suspended state. With the CB3CD in the Suspended state and its service description not included for Advertisement, the Discovery Reply message sent from the global service registry will not contain the service description of services offered by CB3CD.

3.3.6 Notify

Notify is a SOA functionality used by the global service registry to inform service consumers that the service descriptions stored in the global service registry are changed, which reflects that new services are advertised or existing services are removed. Referring to Figure 3.2, Notify
message is sent (broadcasted) when new services are advertised to or existing services are removed from global service registry (which can occur due to updated advertisement from service provider containing changed service descriptions, or after the expiry of a particular advertisement that leads to the removal of service descriptions associated to the advertisement from the global service registry). The structure of Notify message is shown by an example in Listing 3.11.

Listing 3.11. Example of Notify message.

```json
1 { 
2   "msgType" : "Notify",
3   "regID" : "registry1",
4   "notifyID" : "574847038"
5 }
```

Referring to Listing 3.11, Notify message has three JSON keys. The ‘msgType’ key (line 2) defines the type of the message, which is ‘Notify’ referring to Notify. The ‘regID’ key (line 3) defines the identifier of the global service registry that sends the Notify message. The ‘notifyID’ key (line 4) defines the identifier of the Notify message. Each occasion of Notify transmission uses different notifyID. This message identifier is necessary for receiving service consumer to be able to identify when there is a new Notify message sent by the global service registry.

### 3.4 Service Invocation

Once the service consumer side performs Discovery and receives Discovery Reply, it may perform service matching to find particular services which match its requirements based on the obtained service description. Note that SOSJ gives the liberty and flexibility to the programmers to implement their own mechanisms to perform service matching based on their application’s requirements (application-dependent). Once the service matching process is completed, the service invocation process can be initiated. Service invocation in SOSJ follows the SystemJ communication semantics, i.e., reactions invoke services provided by other reactions in the same CD or in different SOSJ programs via signal, while reactions invoke services provided by other reactions in different CDs that belong to the same SOSJ program via channel.

Service invocation via signal is relatively simple and straightforward compared to service invocation done via channel. Once the signal is emitted and becomes present, it will be visible to all reactions in the same CD. Therefore, there is no need for any reconfiguration process to achieve dynamic binding between the service consumer and service provider for the service consumer to invoke services provided by different service providers (in this case, different reactions with service provider role in the same CD) via signal. Service invocation via signal is also applicable
SOSJ: The Integration of SOA Paradigm with SystemJ

between CDs in different SOSJ programs. Explanation regarding how service invocation using signal is achieved will be elaborated in Section 5.2.3. On the other hand, achieving dynamic binding via channel (service invocation via channel) requires dynamic pairing of channel ports during runtime, meanwhile the pairing channel ports in SystemJ is determined during design time and remains fixed during runtime.

To realize dynamic binding of channel ports, the original SystemJ channel semantics are revisited to identify what needs to be changed in the System channel semantics (if any) and to what extent.

Being based on CSP communication semantics, SystemJ channel has the following requirements and properties, as elaborated in [172]:

**Requirement 1**: Channels always communicate in pairs. Hence, for a given output channel named “g” there must exist an input channel also named “g”. These channel pairs can only be engaged in rendezvous once in any given SystemJ system at any instant of time. However they can be reused after completion or preemption of the ongoing rendezvous attempt. Using send and receive on the same channel pair in synchronous concurrent reactions is prohibited.

**Requirement 2**: An output/input channel pair named “g” can only be declared once in the whole system.

**Requirement 3**: A pair of channels adhere to simplex communication protocol with regards to data, i.e., data can only be sent one way from the output channel to the corresponding input channel.

**Property 1**: Rendezvous communications are point-to-point communications between a pair of clock-domains through uni-directional channels. Clock-domains have the ability to pre-empt an ongoing or uninitialised rendezvous. They also have the ability to commit to multiple rendezvous’ concurrently, with any of the partner domains using explicitly different channels.

**Property 2**: All communication between clock-domains is through channels that takes finite and non-zero time to travel from the source domain to the destination domain.

**Property 3**: Given a deadlock free SystemJ program, a clock-domain will never deadlock on a rendezvous attempt due to strong preemptions.

By looking at the aforementioned requirements and properties of SystemJ channel, it can be inferred that all of the requirements and properties of channel don’t need to be changed in order to
achieve dynamic binding between input and output channel ports in SOSJ, with the exception of Requirement 1. In Requirement 1, it is stated that “…for a given output channel named “g” there must exist an input channel also named “g”…” While this particular requirement is meant to ensure that the all channel communications in a SystemJ program are analysable during design time, this enforces channel port pairing to be constrained on only channel ports with the same name. Since channels (as service interfaces) of different CDs may have different names, this requirement hinders the possibility of having dynamic binding of channel ports. Dynamic binding of channel ports is needed when, e.g., CDs with service consumer role invoke services provided by different CDs dynamically. Hence, in SOSJ the aforementioned part in Requirement 1 is disregarded, thus allowing input and output channels with different name to be paired during runtime to achieve dynamic binding. Meanwhile, the rest of the channel properties and communication semantics are retained. Despite that the proposed change in channel semantics (i.e., the introduction of the pairing of input and output channel ports with different name) can be considered as relatively minor, this change is crucial in order to allow dynamic binding of channel (for service invocation in particular) during runtime.

Furthermore, there are additional considerations with regard to enabling dynamic binding of channel in SOSJ. According to the nature of service invocation in SOA, a particular service may be invoked by one or more service consumers during runtime, and also, a service consumer may invoke one or more services provided by one or more providers during runtime. This leads to the possibility in SOA that the interaction between service consumer and provider may become one-to-many and vice-versa (in nature) during runtime. However, the channel communication semantics in SystemJ enforces point to point, one-to-one communication only. With the proposed channel communication semantics, in SOSJ, channel binding between provider reaction and consumer reaction’ has to follow the same one-to-one, point-to-point rule (similar to SystemJ), however the binding (coupling) of channel input and output ports can change during runtime, to allow one service consumer to invoke multiple different services offered by different service providers and also one service to be invoked by multiple service consumers. This new requirement is proposed and denoted as Requirement 4 as follows.

**Requirement 4**: The binding of input and output channel ports remains a one-to-one connection, however the binding is not static and may change during runtime.

---

7 In case of service invocation via channel, both consumer and provider reactions have to reside in different CDs to use channel.
SOSJ : The Integration of SOA Paradigm with SystemJ

Due to dynamic binding and the possibility of one-to-many and many-to-one interaction between provider reactions and consumer reactions during runtime, it is possible that in SOSJ, one or more input channel ports may not be paired with any partner output channel ports and vice-versa due to the one-to-one channel communication rule. However, despite that SystemJ channel communication semantics assume that each channel port in a SystemJ program has a partner, the rest of the channel communication semantics (besides Requirement 1 that is no longer applicable) is not altered, thus it is possible for channel ports not to have any partner channel ports. In case of channel communication where the input or output channel port doesn’t have any partners (unpaired), the channel communication blocks until the corresponding input or output channel port is paired. Once the corresponding input or output channel port is paired with a partner channel port during runtime, the channel rendezvous will be completed after a finite number of ticks, assuming that both channels remain paired until the rendezvous is completed.

In addition to Requirement 4, the following property also needs to be introduced with regard to channel reconfiguration to achieve dynamic coupling/pairing of input and output channel ports during runtime, while at the same time ensuring that the rendezvous of the paired input and output channel ports is not interrupted by attempts from other output channel ports to pair with the corresponding input channel port of the pair due to the one-to-many and many-to-one nature.

**Property 4:** The point from when the request for channel reconfiguration (for transmitting service invocation request message) is acknowledged until the channel communication (the transmission of the service invocation request message) between the consumer and provider is completed is considered as a critical section (“occupied”)

Channel reconfiguration process for service invocation has to be performed properly and safely to ensure that the coupling between the output channel port used by the consumer reaction and the input channel port used by the provider reaction is established properly. Since channel communication may consume more than one tick to finish and must be completed without any interruptions (e.g., when a CD with service consumer role wishes to invoke a service offered by another CD with service provider role, however the only input channel of the CD with service provider role is currently blocking and hasn’t finished its rendezvous with an output channel of another CD with service consumer role), a new rule for channel reconfiguration has to be

---

*Typically both service consumer and service provider will remain paired in the context of SOA-based systems at least until the service consumer has obtained results generated by services offered by service provider, which is then the service consumer may detach from the service provider to invoke other services offered by different service providers*
introduced. By taking these factors into consideration, the new rule in channel reconfiguration process for service invocation is proposed [173] and follows the process flow shown in Figure 3.3.

Referring to Figure 3.3, to initiate channel reconfiguration, the consumer side has to make a request to reconfigure its output channel port used to invoke the service and the input channel port of the provider side which will accept the service invocation request. For every execution cycle of all CDs in the subsystem, only one request to reconfigure the same input channel of a particular provider will be handled on a first come first served basis. Thus, if a request (for example, referred as ReqA) to reconfigure the input channel from one output channel is made while a request (for example, referred as ReqB) to reconfigure the same input channel from another output channel has been received, ReqA is dropped by the RTS and it’s up to the requester (of ReqA) whether to put another request for channel reconfiguration in the future.

![Figure 3.3. Channel reconfiguration for service invocation rule.](image-url)

If a request to reconfigure the input channel is made when there is no other request to reconfigure the input channel, a check is performed to determine whether the partner channel is located in the same subsystem or in another subsystem. If the partner channel is located in the same subsystem, another check is performed whether the partner channel is in the ‘occupied’
status. If the partner channel is ‘occupied’, the channel reconfiguration process terminates and the consumer side can query for the status (Return Fail). Otherwise, both the output channel used by the consumer and the input channel used by the provider are reconfigured and the consumer side can query for the status that the channel reconfiguration is successful (Returns OK). If the partner channel port is located in another subsystem, a check is performed whether there are any existing links that connect the two subsystems. If no existing link is present, a link creation process is initiated (which will be detailed in Section 6.2.1). Once the link is created, the partner subsystem is queried whether the partner channel port is in the ‘occupied’ status. If the partner channel is in the ‘occupied’ status, the channel reconfiguration process is terminated and the consumer side is informed so it can make a new request for channel reconfiguration (Return Fail). If the partner channel is not in the ‘occupied’ status, the output channel of the consumer is reconfigured and a request is sent to the partner subsystem side to reconfigure the input channel of the provider via link. Upon the completion of the above process, the service consumer side is informed that the channel reconfiguration process is successful and that service invocation process can proceed. In the current implementation of the SOSJ, channels are reconfigured by the RTS during the housekeeping time (execution of reactive interface by the SOSJ Run-Time Support, RTS) of the last CD to be executed in the subsystem. Figure 3.4 shows a timing diagram which pinpoints when the reconfiguration occurs (referred as the reconfiguration time). The ET(CD) denotes the execution time of CD, while HK denotes the housekeeping time.

![Timing Diagram](image)

Figure 3.4. Timing diagram which pinpoints when reconfiguration occurs.

New programming constructs and mechanisms which handle service invocation are implemented as part of the SOSJ RTS. Their description and how they are used will be elaborated in detailed in Section 5.2.3.

### 3.5 Concluding Remarks

This chapter has proposed a novel programming paradigm, called SOSJ, which fuses service oriented approach, SOA, with formal model of computation based approach of SystemJ for handling distributed software systems. The chapter has also described a new view of software behaviours based on the integration of the two. New SOA functionalities, service description, and service registry have been proposed to achieve the SOA-based information exchange mechanisms.
and to allow service consumers to find and invoke services offered by service providers. Mechanisms to invoke service (service invocation) have been described. The next chapter (Chapter 4) will describe further extensions, which will provide support for a wider range of dynamic behaviours and scenarios.
4 Extending SOSJ for Dynamic Behaviours

The integration of the SOA paradigm with SystemJ introduces the SOA-based information exchange mechanisms (based on Figure 2.1 and Figure 3.2) and dynamic coupling between CDs for enabling dynamic orchestration of software behaviours. However, the SOA paradigm doesn’t specifically define particular solutions to enable dynamic behaviours, i.e., dynamic creation, suspension, resumption, termination, and migration of software services, which address the dynamicity requirement. Therefore, additional provisions are introduced to SOSJ in order to support full dynamic behaviours.

The new features for dynamic reconfiguration extend the static GALS model of SystemJ into dynamic GALS, which allows a program to be described at any point in time with fixed but changeable number of CDs. Dynamic GALS, which is not formally defined in this thesis, can be informally considered as a MoC that has state defined by the number of CDs executing at any point in time. As new CDs are introduced or others removed from the program dynamically, the state of the program changes, but within each state the program behaves as a static GALS program. Thus, the semantics of program in any state remains as defined in the original SystemJ. Changes/transitions of the states happen outside of the execution time of affected (those that join or exit the program) CDs, i.e., in the housekeeping phase when the RTS performs functions of reactive interface and all operations related to dynamic features and reconfiguration in SOSJ. This must be carried out at the end of logical tick of affected CDs as logical tick computations of a CD are considered atomic (tick must complete).

Also, the SystemJ CD semantics define that CD is always executed in every tick during runtime. However, when scenarios of dynamic behaviours are considered, the existing CD semantics are not sufficient to cover these scenarios. This chapter proposes CD macro-states and their transitions. The CD macro-states and their transitions extend the existing CD semantics to
Extending SOSJ for Dynamic Behaviours
cater for scenarios of dynamic behaviours. They need to be formally described for the dynamic behaviour handling functionalities of SOSJ. In addition, dynamic behaviours can involve changes in interactions between CDs, modifications of CD signal and channel mapping, and creation of links during runtime, thus the configuration of CD, their signal and channel mapping, and the physical location of subsystem need to be retained during runtime and become parts of the SOSJ RTS. This chapter describes the aforementioned CD and subsystem configuration (which includes the CD signal and channel mapping) and how they are included as parts of the SOSJ RTS.

The chapter is organized as follows. Section 4.1 describes the entire CD life cycle formed through the transitions of CD macro-states for all possible dynamic behaviour scenarios. Next, Section 4.2 introduces CD and subsystem configuration as a data structure which describes which subsystem CD belongs to, which macro-state CD is currently in, the configuration of CD’s signals and channels, and also the physical location (address) of subsystem. Finally, Section 4.3 describes the transitions of CD macro-state which occur due to dynamic behaviours.

4.1 Introducing CD Macro-states to SOSJ

Since SystemJ is originally catered for designing static distributed systems, the semantics of CD in SystemJ define that SystemJ CD runs in every tick once the execution starts. However, this is not sufficient when scenarios of dynamic behaviours are considered. Thus, the semantics of CD need to be extended to cater for dynamic behaviours by introducing CD macro-states and their transitions. Note that as previously mentioned in Section 3.1, reactions in SOSJ are not considered as the equivalent of service entities in SOA. Similarly, with regard to granularity of dynamic behaviours, dynamic behaviours in SOSJ are not realized at the reaction-level since reactions in the same CD are synchronous with each other, and therefore, they are bound to the CD. The compliance of SOSJ with the GALS MoC, and particularly the SR MoC within one CD, makes the introduction of dynamic behaviours handling at reaction level unnecessary. In addition, it is likely that the feature of functional correctness in SOSJ will be compromised if dynamic behaviours are introduced at the reaction level. Thus, dynamic behaviours in SOSJ are realized at the CD-level.

Each CD has CD descriptor, a Java ‘object’ created from the CD code class file which contains the execution, data, and communication (signal and channel) state of the CD. When a CD is run, the execution, data, and communication (signal and channel) state of the CD are updated and stored
Extending SOSJ for Dynamic Behaviours

in the CD descriptor during runtime. Apart from the execution, data, and communication state, CD descriptor also stores the CD’s name and its present macro-state.

The transition of CD macro-state can be triggered by CDs or events from the SOSJ RTS. Figure 4.1 illustrates all CD macro-states and their transitions [58]. The description of each CD macro-state is as follows:

**Dormant**: In this state, the CD is present in the memory of the underlying machine, however the CD is not yet schedulable. The CD descriptor, the CD’s signals and channels, and service description are created to prepare the CD to move into the Active state. Services offered by the CD (if any) in the Dormant state are not yet advertised.

**Active**: Once the CD descriptor, along with its associated signals and channels and service descriptions, are created, the CD becomes schedulable and is considered to be in the Active state. The Active state consists of three sub-states, which are Ready, Running, and Suspended. The Ready state represents the CD which has been created and is available for scheduling. The Running state is when the CD is included in the scheduler. In this state, services offered by the CD (if any) are advertised and thus become available to consumer reactions. A CD is considered in the Suspended state when it is temporarily excluded from scheduling. Note that while CDs in one subsystem may be scheduled using different kinds of scheduling policies, SOSJ assumes the use the round-robin (cyclic) scheduling policy[^10], which is the default CD scheduling policy used in SystemJ. The CD state of execution, communication (i.e. statuses and values of signals and channels), and data are maintained to enable resumption of the CD execution. Services description of the CD is retained, however, its services are no longer advertised and become temporarily

[^10]: The use of CD scheduling policies other than the default round-robin scheduling policy in SOSJ is out of the scope of this research.
Extending SOSJ for Dynamic Behaviours

unavailable to clients. When the CD is to be migrated, it is brought into the **Suspended** state to retain all the aforementioned statuses needed for the resumption of the execution of the CD after migrating to a new location.

**Migrating:** CD in this state is currently being transferred from the original location into a new location. The CD remains in this state until the transfer is completed. There are two types of migration which are supported by SOSJ, weak and strong migration. Strong migration allows CD mobility that retains the CD’s data, execution, and communication state and also the CD’s macro-state prior to transitioning to the Migrating state (whether the CD was in the Running or Suspended state before entering the Migrating state\(^\text{11}\)), while weak migration enables CD mobility without having the state of the CD retained. Strong migration involves the transfer of the CD code, CD descriptor, configuration (including the CD signal and channel mapping, explained in Section 4.2), and service description (if any), while weak migration involves all of the above except the CD descriptor. Note that all elements of the CD that have migrated (i.e. the CD descriptor, code, CD configuration, service description) are erased from the memory in the original location since the CD has moved to a different location. In the context of SOSJ, CD migrates from one subsystem to another subsystem.

**Terminated:** When the CD is completely erased from the memory, it is considered to be in the Terminated state. Once reaching this state, the service(s) offered by the CD are permanently unavailable.

Let \( S S_1 \) be the corresponding subsystem named subsystem \( I \), \( Adv_1 \) be the set containing all service descriptions of services associated to all CDs in subsystem \( I \) which are included for advertisement, \( D_1 \) be the set which contains all CD descriptor of CDs in subsystem \( I \), \( S_1 \) and \( Ch_1 \) be the sets which contain all signals and channels used by all CDs in subsystem \( I \), respectively. Let \( Sc_1 \) be the set which denotes all CDs in subsystem \( I \) included in the scheduler handled by the RTS of subsystem \( I \).

Based on the aforementioned notations, the semantics of the aforementioned CD macro-states can be described as in Expression 4.1 - 4.9. Note that \( CD_1 \) denotes the corresponding CD, \( SD_1 \) denotes the service description associated to \( CD_1 \), \( D_1 \) denotes the CD descriptor of \( CD_1 \), \( S_1 \) denotes the set that contains all signals used by \( CD_1 \), \( Ch_1 \) denotes the set that contains all channels used by \( CD_1 \), \( MCD_1 \) denotes the set containing the migrating CDs originally associated to subsystem \( I \)

\(^{11}\) This means that ‘in Running state’ CD which undergoes strong migration will be in the Running state at the new location once the migration process is finished, and similarly, ‘in Suspended state’ CD which undergoes weak migration will be in the Suspended state at the new location once the migration process is finished.
which are being transferred and queued for transfer to another subsystem for migration (in the Migrating state), \( MD_I \) denotes the set containing all CD descriptors of the migrating CDs originally associated to subsystem \( I \) which are being transferred and queued for transfer to another subsystem for migration (i.e. in the Migrating state), \( DS_I \) denotes the set containing all CD descriptors which are stored to retain the overall state of CDs in the Suspended state, \( SS_J \) denotes the migration destination subsystem named subsystem \( J \), \( Adv_J \) denotes the set containing all service descriptions of services associated to all CD in subsystem \( J \) which are included for advertisement, \( D_I \) denotes the set which contains all CD descriptor of CDs in subsystem \( J \), \( S_I \) and \( Ch_I \) denote the sets which contain all signals and channels used by all CDs in subsystem \( J \) (respectively), \( Sc_I \) denotes the set which denotes all CDs in subsystem \( J \) included in the scheduler handled by the RTS of subsystem \( J \), \( ACD_J \) denotes the set containing the migrating CDs which are currently transferred and the ones which have been transferred to subsystem \( J \), \( AD_J \) denotes the set containing all CD descriptors of the migrating CDs which are currently transferred and the ones which have been transferred to subsystem \( J \).

Expression (4.1) – (4.5) represent the Dormant, Ready, Active, Suspended, and Terminated state, respectively. The Migrating state is defined in Expression (4.6) – (4.9). Expression (4.6) specifies the Migrating state at the origin subsystem in case of strong migration, Expression (4.7) specifies the Migrating state at the origin subsystem in case of weak migration, Expression (4.8) specifies the Migrating state at the destination subsystem in case of strong migration, and Expression (4.9) specifies the Migrating state at the destination subsystem in case of weak migration. The difference between both migrations is that the strong migration will transfer the CD descriptor of the corresponding CD \( CD_I \) \((D'_I)\) to retain execution, data, and communication (signal and channel) states for the migrating \( CD_I \) in the new location (in this context, the destination subsystem), while the weak migration doesn’t include the CD descriptor of the CD \( CD_I \) \((D'_I)\) for transfer since weak migration doesn’t retain execution, data, and communication states of the CD \( CD_I \):
Extending SOSJ for Dynamic Behaviours

\[
\begin{align*}
CD_{i'} \notin SS_j, CD_{i'} \notin Sc_j, SD_{i'} \notin Adv_j, D_{i'} \notin D_j, S_{i'} \notin S_j, Ch_{i'} \notin Ch_j, CD_{i'} \notin MCD_j, \\
D_{i'} \notin MD_j & \quad (4.7) \\
CD_{i'} \notin SS_j, CD_{i'} \notin Sc_j, SD_{i'} \notin Adv_j, D_{i'} \notin D_j, S_{i'} \notin S_j, Ch_{i'} \notin Ch_j, CD_{i'} \notin ACD_j, \\
D_{i'} \notin AD_j & \quad (4.8) \\
CD_{i'} \notin SS_j, CD_{i'} \notin Sc_j, SD_{i'} \notin Adv_j, D_{i'} \notin D_j, S_{i'} \notin S_j, Ch_{i'} \notin Ch_j, CD_{i'} \notin ACD_j, \\
D_{i'} \notin AD_j & \quad (4.9)
\end{align*}
\]

### 4.2 CD and Subsystem Configuration in SOSJ

To facilitate the reconfiguration of (environment) signals and channels following dynamic behaviours of CD, new data structures are introduced to store the CD and subsystem configuration, the association of CDs with their subsystems, the macro-state of CDs, and the physical location (address) of subsystems.

Similar to service description, the CD and subsystem configuration is written by programmers in XML format. During runtime, it is converted to JSON format and stored in the SOSJ RTS. Note that the CD and subsystem configuration used by SOSJ is similar with the one in SystemJ, except that it has additional XML elements and attributes (compared to the CD and subsystem configuration in SystemJ) which will be elaborated in the next paragraph. Listing 4.1 shows an example of a CD and subsystem configuration to illustrate how it is structured and written in XML format by programmers, while Listing 4.2 shows how the CD and subsystem configuration example in Listing 4.1 is mapped into JSON format and stored in the RTS.

**Listing 4.1. Example of CD and subsystem configuration written in XML.**

```xml
1  <SubSystem Name="SS2" SSAddr="192.168.1.10" />
2  <ClockDomain Name="D1CD" Class="D1CDClass" >
3      <Services>
4          <!--…service description…//
5          </Services>
6      <SignalChannel>
7          <iChannel Name="D1ReqCh1" From="." />
8          <Signal Name="DPosSens1" Class="DiverterIn" Platform="BB" GPIOPinNum="1" />
9          <oSignal Name="actDiv1" Class="DiverterControl" Platform="BB" GPIOPinNum="11" />
10         <oChannel Name="D1OChan1" To="CB4CD.CB4InChan1" />
11      </SignalChannel>
12  </ClockDomain>
13  <ClockDomain Name="CB4CD" Class="CB4CDClass" >
14      <Services>
15          <!--…service description…//
16          </Services>
17      <SignalChannel>
18      </ClockDomain>
```

76
Extending SOSJ for Dynamic Behaviours

Referring to Listing 4.1 and Listing 4.2, the configuration describes the configuration of CDs that belong to a subsystem named SS2, which is the same subsystem described in the example shown in Listing 3.3. The configuration in Listing 4.1 includes the configuration of three CDs, namely D1CD, CB4CD, and PE1CD, which govern the behaviour of diverter D1, conveyor CB4, and photo eye sensor PE1 of the motivating example in Figure 1.1, respectively. The CD and subsystem configuration has a number of XML elements and attributes. The Subsystem element (line 1 of Listing 4.1) contains two attributes, Name and SSAddr. The Name attribute defines the subsystem name or identifier (mapped to line 1 of Listing 4.2). The SSAddr attribute defines the physical location of the subsystem, which can be an IP address, URI, etc. In the example, subsystem SS2 is deployed on a machine which has an IP address of 192.168.1.10. The physical location of the subsystem defined by the SSAddr attribute is mapped and stored in another data structure (generated by the SOSJ framework automatically) shown in Listing 4.3, used to store the information regarding the physical locations of individual subsystems. The subsystem identifier (as defined in the Name attribute of the SubSystem element in line 1 of Listing 4.1) and the corresponding physical location of the subsystem (as defined in the SSAddr attribute of the SubSystem element in line 1 of Listing 4.1) are both mapped to line 2 of Listing 4.3.

Listing 4.2. Mapping of CD and subsystem configuration to JSON format.
Extending SOSJ for Dynamic Behaviours

```
"D1OChan1": {
    "To": "CB4CD.CB4InChan1"
},
"input": {
    "D1ReqCh1": {
        "From": "."
    }
},
"Signal": {
    "output": {
        "actDiv1": {
            "Class": "DiverterControl",
            "Platform": "BB",
            "GPIOPinNum": "11"
        }
    },
    "input": {
        "DPosSens1": {
            "Class": "DiverterIn",
            "Platform": "BB",
            "GPIOPinNum": "1"
        }
    }
},
"CB4CD": {
    "CDClassName": "CB4CDClass",
    "Channel": {
        "input": {
            "CB4InChan1": {
                "From": "D1CD.D1OChan1"
            }
        },
        "output": {
            "CB4RespCh1": {
                "To": "."
            }
        }
    },
    "Signal": {
        "input": {
            "speedSens4": {
                "Class": "ConveyorIn",
                "Platform": "BB",
                "GPIOPinNum": "2"
            }
        },
        "output": {
            "actConv4": {
                "Class": "ConveyorControl",
                "Platform": "BB",
                "GPIOPinNum": "12"
            }
        }
    }
}
```
Extending SOSJ for Dynamic Behaviours

```
  "PE1CD": {
    "CDClassName": "PE1CDClass",
    "Channel": {
      "output": {
        "D1RespCh1": {
          "To": "."
        },
        "input": {
          "PE1ReqCh1": {
            "From": "."
          }
        }
      },
      "Signal": {
        "input": {
          "peSens1": {
            "Class": "PESensor",
            "Platform": "BB",
            "GPIOPinNum": "13"
          }
        }
      }
    }
  }
```

Listing 4.3. Physical location of subsystem data structure.

```
  "SS2": "192.168.1.10",
}
```

The configurations of D1CD, CB4CD, and PE1CD are described in line 2-13, line 14-25, and line 26-35 of Listing 4.1 respectively. Their configurations are mapped to line 2-36, line 37-71, and line 72-96 of Listing 4.2, respectively. The configuration of CD begins with the ClockDomain elements (line 2, line 14, and line 26 of Listing 4.1). The ClockDomain has two attributes, Name and Class. The Name attribute defines the CD name (identifier), while the Class attribute refers to the Java class file name of the CD code. The CD names defined by the Name attribute of the ClockDomain element are mapped to line 2, line 37, and line 72 of Listing 4.2, respectively, while the Java class file name of the CD code defined by the Class attribute of the ClockDomain element are mapped to line 2, line 38, and line 73 of Listing 4.2, respectively. The Services element (line 3, line 15, and line 27 of Listing 4.1) encapsulates the service description of services associated to
Extending SOSJ for Dynamic Behaviours

the CDs that belong to the subsystem. Note that in SOSJ, the CD and subsystem configuration is put together with the service description (with structure shown in Listing 3.1, mapped to a JSON format with structure shown in Listing 3.2) so they will be parsed together by the SOSJ RTS right after the subsystem begins its execution.

The configuration of signals and channels used by individual CDs is encapsulated inside the SignalChannel element (line 6-12, line 18-24, line 30-34 of Listing 4.1). Note that for signals, the configuration specifies only signals which are used to interact with the physical environment (or interface signals). Unlike the interface signals, signals which are used by reactions to communicate with other reactions in the same CD (or internal signals) are not included in the CD and subsystem configuration.

The configuration of an input interface signal begins with an iSignal element. The iSignal element has at least two attributes, Name and Class. The Name attribute defines the name of the input signal, while the Class attribute defines the Java class file which describes how the physical interface used by the input interface signal is implemented. For example, line 8 of Listing 4.1 shows the configuration of an input interface signal named DPosSens1 which obtains the diverter D1’s position whether the diverter D1 is currently in opened or closed position. The DPosSens1 input signal uses a Java class file named Diverter that provides the implementation of the physical interface used by the input signal. Depending on the nature of the signal’s physical interface and how it is implemented in the Java class file handling the physical interface, additional attributes may be required besides the Name and Class attributes. For the DPosSens1 signal, the Diverter class file needs additional parameters defined in the Platform and GPIOPinNum attributes, also in line 8 of Listing 4.1. In this case, the Platform attribute defines the underlying physical machine where the CD executes, while the GPIOPinNum attribute defines the physical input and output (I/O) pin mapping number of the underlying physical machine which is used to obtain the physical signal from the position sensor of the diverter D1. According to the values defined by the Platform and GPIOPinNum, the signal is configured to use the physical I/O pins of the Beaglebone Black (BB) execution platform [174]. The DPosSens1 input signal configuration in line 8 of Listing 4.1 is mapped to line 28-32 of Listing 4.2.

On the other hand, the configuration of an output signal starts with an oSignal element. Similar to the iSignal element, oSignal element has at least two attributes, Name and Class attributes. The Name attribute defines the name of the output signal, while the Class attribute defines the Java class file which describes how the physical interface used by the output interface signal is implemented. For example, line 9 of Listing 4.1 shows the configuration of an output
Extending SOSJ for Dynamic Behaviours

interface signal named \textit{actDiv1} which actuates the diverter D1 from the opened position to closed position or vice-versa. The \textit{actDiv1} output signal uses a Java class file named \textit{DiverterControl} that provides the implementation of the physical interface used by the output signal. Similar to the \textit{DiverterIn} class used by the \textit{DPosSens1} signal, the \textit{DiverterControl} class used by the \textit{actDiv1} signal needs additional parameters defined in the \textit{Platform} and \textit{GPIOPinNum} attributes, also in line 9 of Listing 4.1. Similar to the ones found in line 8 of Listing 4.1, the \textit{Platform} attribute defines the underlying physical machine where the CD executes, while the \textit{GPIOPinNum} attribute defines the physical input and output (I/O) pin mapping number of the underlying physical machine which is used to provide the physical control signal to actuate diverter D1. Similar to the one in \textit{iSignal}, according to the values defined by the \textit{Platform} and \textit{GPIOPinNum} attributes, the signal is configured to use the physical I/O pins of the Beaglebone Black (BB) platform. The \textit{actDiv1} output signal configuration in line 9 of Listing 4.1 is mapped to line 21-25 of Listing 4.2.

Besides interface signals, both input and output channels configuration are also stored in the CD and subsystem configuration. The configuration of an input channel starts with the \textit{iChannel} element, while the configuration of an output channel begins with the \textit{oChannel} element. Compared to both \textit{iSignal} and \textit{oSignal} elements, the \textit{iChannel} and \textit{oChannel} elements have only two attributes. The \textit{iChannel} element has the \textit{Name} and \textit{From} attributes, while the \textit{oChannel} element has the \textit{Name} and \textit{To} attributes. For both the \textit{iChannel} and \textit{oChannel} elements, the \textit{Name} attribute defines the name of the input or output channel ports. In case of \textit{iChannel} element, the \textit{From} attribute defines the partner output channel of the corresponding input channel, while in case of \textit{oChannel} element, the \textit{To} attribute defines the partner input channel of the corresponding output channel. An example on how input channel is configured is shown in line 7 and line 19 of Listing 4.1. In line 19 of Listing 4.1, the configuration of an input channel named CB4InChan1 is presented. The \textit{From} attribute of the input channel CB4InChan1 is defined as “D1CD.D1OChan1”. There are two elements in the parameter defined by the \textit{From} attribute, separated by a dot (”.”) sign. The first element refers to the CD which uses the partner output channel (“D1CD”), while the second element refers to the name of the partner output channel (“D1OChan1”). The configuration in line 19 of Listing 4.1 is mapped to line 41-43 of Listing 4.2. In line 7 of Listing 4.1, the configuration of an input channel named D1ReqCh1 is presented. However, different from the input channel CB4InChan1, the \textit{From} attribute of the input channel D1ReqCh1 is defined as “.” to indicate that the partner output channel of the input channel D1ReqCh1 is initially left unpaired and is not determined during design time. Thus, the input channel D1ReqCh1 can be paired with any output channels used by CDs other than the D1CD CD.
Extending SOSJ for Dynamic Behaviours during runtime. This method of configuring input channel can be applied when the input channel is used as a service interface function for the D1CD to pair with output channels of any consumer CDs to accept service invocation message from consumer CDs where the invoking service consumers may change dynamically during runtime. The configuration in line 7 of Listing 4.1 is mapped to line 14-16 of Listing 4.2.

An example of how output channel is configured is shown in line 11 and line 23 of Listing 4.1. In line 11 of Listing 4.1, the configuration of an output channel named D1OChan1 is presented. The To attribute of the output channel D1OChan1 is defined as “CB4CD.CB4InChan1”. Similar to the From attribute of the iChannel element, there are two elements in the parameter defined by the To attribute of the oChannel element, separated by a dot (".") sign. The first element refers to the CD which uses the partner input channel (“CB4CD”), while the second element refers to the name of the partner input channel (“D1OChan1”). The configuration in line 11 of Listing 4.1 is mapped to line 9-11 of Listing 4.2. In line 23 of Listing 4.1, the configuration of an output channel named CB4RespCh1 is presented. Similar to the input channel D1ReqCh1, the output channel CB4RespCh1 is left unpaired during design time by having the To attribute of the output channel’s configuration in line 23 of Listing 4.1 defined as “.”. This method of configuring output channel can be applied when the output channel is used to send service results to the invoking service consumers where the invoking service consumers may also change dynamically during runtime. The configuration shown in line 23 of Listing 4.1 is mapped to line 49-51 of Listing 4.2.

Furthermore, the SOSJ framework generates additional data structures that contain the association of CDs with their subsystems, the macro-state of CDs, and the physical location of subsystems automatically during runtime automatically based on the content of the CD and subsystem configuration. The first additional data structure is used to store information regarding the associated of CDs to the subsystem. This data structure is crucial for the purpose of link creation and channel reconfiguration process (explained in Section 6.2.1 and 6.2.2). An example of this data structure is shown in Listing 4.4 with the individual CDs belonging to the subsystem SS2 which have the configuration described in Listing 4.1.

Listing 4.4. Example of data structure of the association of CDs to the subsystem.

```
{ 
    "CB4CD": "SS2", 
    "PE1CD": "SS2", 
    "D1CD": "SS2" 
}
```
Extending SOSJ for Dynamic Behaviours

As shown in Listing 4.4, the names of the individual CDs as defined by the CD and subsystem configuration (line 2, line 14, and line 26 of Listing 4.1) are stored as keys in the data structure, while the subsystem name where the individual CDs belong to (line 1 of Listing 4.1) is stored as a value in the data structure (line 2, line 3, and line 4 of Listing 4.4).

The second additional data structure stores the macro-state of individual CDs in the subsystem. This data structure will be used by the SOSJ RTS for dynamic behaviour handling functionalities described in Section 6.2. An example of this data structure is shown in Listing 4.5. Similar to Listing 4.4, the example considers the individual CDs belonging to the subsystem SS2 that have the configuration described in Listing 4.1. As shown in Listing 4.5, the names of the individual CDs as defined by the CD and subsystem configuration in line 2, line 14, and line 26 of Listing 4.1 are stored as the keys in the data structure, while the macro-state of the corresponding CDs are stored as the values in the data structure (line 2, line 3, and line 4 of Listing 4.5).

Listing 4.5. Example of CD macro-state data structure.

```
1 {  
2   "CB4CD" : "Running",
3   "PE1CD" : "Running",
4   "D1CD" : "Suspended"
5 }
```

4.3 CD Macro-state Transitions Due to Dynamic Behaviours

Due to dynamic behaviours of CD, which include dynamic creation, suspension, resumption, termination, and migration (both strong and weak) of CD, the macro-state of the corresponding CD may change during runtime. The transitions of CD macro-state due to the aforementioned dynamic behaviours are explained in the following subsections. The programming constructs and mechanisms available in SOSJ which allow programmers to perform dynamic behaviours will be elaborated in detail in Section 6.2.

4.3.1 CD Creation

In SOSJ, dynamic creation of CDs involves creating new CDs and re-creating existing CDs, e.g., for the purpose of modifying (or updating) the behaviour of existing CDs, during runtime. In case of creating new CDs during runtime, the transitions of the macro-state of the CD comply with the transitions as illustrated in Figure 4.2. As shown in Figure 4.2, the CD begins as being in the Dormant state. When dynamic creation of CD process is initiated (for creating new CD), its CD descriptor, signals, and channels are created and configured based on the CD configuration during
Extending SOSJ for Dynamic Behaviours

the house keeping time before it moves to the Ready state. In addition, the CD configuration and service description of the CD are stored before the CD moves to the Ready state. Finally, the CD is included into the scheduler and its service description included for Advertisement (if any) for it to enter the Running state and execute during the execution time.

Figure 4.2. CD macro-state transitions due to dynamic creation of CD (creating new CD).

Meanwhile, in case of modifying (or updating) the behaviour of existing CDs, the CD macro-state transitions follows the transitions illustrated in Figure 4.3. When the dynamic creation of CD process is triggered (for recreating existing CD), the existing CD which can be in either Running or Suspended state during the execution time is brought to the Dormant state during the house keeping time by having its CD descriptor, signal, channels, and also its CD configuration and service description removed. Then, the new CD descriptor, signals, and channels are reinstantiated based on the new CD configuration\textsuperscript{12}. Also, the new CD configuration and service description are stored before the CD is transitioning to the Ready state. Finally, the CD is put back into the Running state after it is included to the scheduler for execution and have its service description included for Advertisement.

Figure 4.3. CD macro-state transitions due to dynamic creation of CD (modify, update existing CD).

\textsuperscript{12}In regards to dynamic creation of CD for the purpose of re-creating existing CDs, it is most often that new CD configuration and service description are needed in accordance to the modifications/changes in the CD behaviour.
4.3.2 CD Suspension

With regard to dynamic suspension of CD, the CD macro-state transitions comply with the transition illustrated in Figure 4.4. Referring to Figure 4.4, dynamic suspension can only be imposed on CD which is in the Running state. When the dynamic suspension of CD is initiated, during the house keeping time the CD is excluded from the scheduling and its service description (if any) is excluded from Advertisement temporarily. The CD descriptor is stored to retain the CD’s execution, communication (i.e. signals and channels’ statuses and values), and data state. Also, due to the nature of physical interface used by input interface signals which is most often implemented as Java threads in continuous loops as explained in [175], all input interface signals’ physical interface implementations of the CD are terminated from execution. After the aforementioned steps are performed, the CD moves to the Suspended state.

![Figure 4.4. CD macro-state transitions due to dynamic suspension of CD.](image)

4.3.3 CD Resumption

The support for dynamic suspension of CD should be accompanied with the support for dynamic resumption of CD. Dynamic resumption of CD complies with the CD macro-state transitions shown in Figure 4.5. Referring to Figure 4.5, dynamic resumption of CD can only proceed when the CD is initially in the Suspended state. When the dynamic suspension of CD is initiated, the stored CD descriptor which retains the CD’s execution, communication (i.e. signals and channels’ statuses and values), and data state is retrieved during the house keeping time. The Java threads which handle the physical interfaces used by input interface signals are re-instantiated based on the stored CD configuration. Once the aforementioned operations are performed, the CD moves to the Ready state. Finally, the CD is included in the scheduler and its service description is included for advertisement, making the CD enters the Running state during the next execution time.
Extending SOSJ for Dynamic Behaviours

Figure 4.5. CD macro-state transitions due to dynamic resumption of CD.

4.3.4 CD Termination

Apart from being able to be created dynamically during runtime, CD can also be permanently terminated and removed during runtime. Dynamic termination of CD complies with the macro-state transitions shown in Figure 4.6.

Figure 4.6. CD macro-state transitions due to dynamic termination of CD.

Referring to Figure 4.6, termination of CD can be performed on CD which is in either the Running or Suspended state. After CD termination is triggered, during the house keeping time the CD is excluded from the scheduler and has its service description removed from Advertisement. Then, the CD descriptor, the CD code, signals and channels, CD configuration, and the service description of services offered by the CD are removed. Also, the Java threads which handle the physical interfaces used by input interface signals are terminated. After the aforementioned are performed, the CD is considered to be in the Terminated state and will cease to exist in the next execution time.
Extending SOSJ for Dynamic Behaviours

4.3.5 CD Migration

As previously mentioned, CD migration in SOSJ allows the mobility of CD, i.e. to move from one subsystem to another subsystem. The migration of CD triggers CD macro-state transitions both in the source subsystem (origin) and the destination subsystem (destination). In addition, both strong and weak migration trigger different CD macro-state transitions in the destination subsystem. When CD migration is initiated, it triggers CD macro-state transitions in the source subsystem as shown in Figure 4.7.

Referring to Figure 4.7, CD migration may occur on CD which is in either the Running or Suspended state. When CD migration is initiated, the CD is put in the Suspended state after the CD is excluded from the scheduling and also its service description excluded from Advertisement if the CD was in the Running state. Then, the transfer process starts to move the CD into the Migrating state. Once the transfer process is completed, the CD moves to the Terminated state by having its CD descriptor, CD code, signals, channels, CD configuration, and the service description of services offered by the CD removed and all Java threads implementing the physical interfaces used by input interface signals are terminated. The migrated CD ceases to exist in the next execution time.

Once the transfer process is initiated, it triggers CD macro-state transitions of the migrating CD at the destination subsystem. Both strong and weak migration have different CD macro-state transitions. In case of strong migration, the CD macro-state transitions in the destination subsystem complies with the transitions as shown in Figure 4.8.

Referring to Figure 4.8, the CD enters the Migrating state in the destination subsystem once the transfer process is initiated. The CD remains in the Migrating state until the transfer process is completed (which may requires multiple ticks to finish). In case of CD which was previously in the Running state prior to its migration, once the transfer process finishes, the CD will have its CD
Extending SOSJ for Dynamic Behaviours
descriptor, CD code, signals, channels, CD configuration, and service description available for it
to go to the Ready state, before being included into the scheduler and having its service description
included for Advertisement to move the CD into the Running state in the next execution time. In
contrast, in case of CD which was previously in the Suspended state prior to its migration, the CD
goes to the Suspended state by having its CD descriptor excluded from the scheduler and also its
service description excluded from Advertisement once the transfer process is completed. Finally,
the CD remains in the Suspended state in the next execution time.

Figure 4.8. CD macro-state transitions due to dynamic migration of CD at the destination
subsystem (strong migration).

In case of weak migration, the initiation of the transfer process also triggers CD macro-state
transitions of the migrating CD at the destination subsystem. The CD macro-state transitions
follow the transitions shown in Figure 4.9. Referring to Figure 4.9, the migrating CD enters the
Migrating state at the destination subsystem once the transfer process begins. The CD remains in
the Migrating state until the transfer process is finished (could also need more than one tick to
finish). When the transfer process is completed, the CD enters the Dormant state and have its CD
descriptor, signals, channels, CD configuration, and service description reinitialised to prepare it
to move to the Ready state. Finally, the CD is included for scheduling and has its service
description included for Advertisement to move the CD into the Running state in the next
execution time.
Extending SOSJ for Dynamic Behaviours

Figure 4.9. CD macro-state transitions due to dynamic migration of CD at the destination subsystem (weak migration).

4.4 Concluding Remarks

This chapter has proposed extensions into the SOSJ paradigm to cater wider range of dynamic scenarios. The chapter described changes in the CD semantics that include the introduction of CD macro-states and their transitions to cater for a wide range of dynamic behaviours. CD and Subsystem Configuration have been also described in this chapter to facilitate dynamic reconfiguration of signals and channels. The next two chapters, Chapter 5 and Chapter 6, will describe the implementation of the SOSJ programming framework which achieves the SOSJ paradigm and the extensions that cater for dynamic behaviours, respectively.
To address the design and programming requirements of DISS, the new SOSJ programming paradigm which combines the SOA paradigm and SystemJ is proposed in Chapter 3. The paradigm introduces new perspectives of software behaviours according to the SOA paradigm and SystemJ. The paradigm introduces new SOA functionalities with the type of interaction shown in Figure 3.2 to enable CD to discover, advertise, and invoke services. The paradigm also describes rules and concepts with regard to performing service invocation dynamically through signal or channel. However, the mechanisms on how the SOA functionalities and the type of interaction shown in Figure 3.2 are not described in the chapter. This chapter describes how they are achieved by extending the SystemJ RTS. The extensions include the development of a centralized service registry (referred as the ‘global’ service registry) which stores the service descriptions of all advertised services and the new features which are implemented as Java threads, new signal classes, and data structures in the SOSJ RTS to handle SOA functionalities (referred as the ‘SOA RTS’). This chapter describes these extensions which become parts of the SOSJ framework.

This chapter is broken down into two sections. Section 5.1 describes the implementation of the global service registry which is implemented separately from the SOSJ RTS, Section 5.2 describes the implementation of the SOA RTS which is part of the SOSJ RTS that handles SOA functionalities in SOSJ and presents the basic ‘template’ on how to describe CD with service provider or consumer role by making use of the new features and function calls provided by the SOA RTS.

### 5.1 SOSJ Global Service Registry

#### 5.1.1 Design and Architecture

The global service registry is a stand-alone application implemented in Java. It consists of data structures (which store service description, CD macro-states, and physical location of subsystems) and functionalities to realize the interaction between the global service registry and
SOA Implementation in SOSJ

both the Advertising Function and Client Function as shown in Figure 3.2. The design and architecture of the global service registry application is shown in Figure 5.1.

![Figure 5.1. SOSJ Global Service Registry architecture.](image)

As shown in Figure 5.1, the global service registry application can be divided into five parts, **RegBootstrap**, **Mailbox**, **GSR Data Structure**, **Reg SOA Manager**, and **SOA Message Generator**. The **RegBootstrap** is part of the global service registry which acts as a bootstrap function to initiate the functionalities in the **Reg SOA Manager**. The **Mailbox** is a dedicated data structure which is used by different components in the global service registry to communicate and exchange information between each other. The **GSR Data Structure** encapsulates four different data structures, the **Global Service Repo**, **CD Macro-state Repo**, **AdvTimeStamp Repo**, and **Subsystem Location Repo**. The **Global Service Repo** stores all service descriptions which are obtained from the Advertisement messages received by the global service registry. The **CD Macro-state Repo** stores the information regarding the macro-state of CDs. The **Subsystem Location Repo** stores the information regarding the physical location of subsystems. The **AdvTimeStamp Repo** stores the time stamp of Advertisement received by the global service registry and also how long the Advertisement is valid (expiry time)\(^\text{13}\).

The **Reg SOA Manager** comprises five functionalities, the **RegSender**, **RegDiscReceiver**, **RegReceiver**, **RAdv**, and **RegExpiryChecker**. The **RegReceiver** is responsible for receiving Advertisement message (depicted as Adv). The **RegDiscReceiver** receives Discovery message (depicted as Disc). The **RegSender** is in charge of sending SOA messages depicted as RegOut,

\(^{13}\) The expiry time follows the value put in the ‘expiryTime’ key on each Advertisement message.
SOA Implementation in SOSJ

which includes Beacon, Notify, and Discovery Reply messages. The RegExpiryChecker periodically checks for the expiry time of all advertisements received by the global service registry. When an Advertisement reaches its expiry, the RegExpiryChecker removes the stored service descriptions, the information about the macro-state of CDs, and the physical location of the subsystem associated with the expired advertisement and triggers the transmission of Request for Advertisement. The RAdv is responsible for transmitting Request For Advertisement (depicted as ReqAdv) and receiving the Advertisement reply messages (depicted as AdvRep). The **SOA Message Generator** provides Java methods which are internally used by the global service registry application to generate SOA messages.

### 5.1.2 Implementation

The global service registry possesses a bootstrap functionality which initiates the execution of functionalities in the Reg SOA Manager. Note that the global service registry application considered in this thesis assumes the use of IPv4-based communication for sending and receiving SOA messages (i.e. Beacon, Discovery, Discovery Reply, Advertisement, and Request for Advertisement). The code snippet showing parts of the RegBootstrap function of SOSJ is shown in Listing 5.1.

#### Listing 5.1. The RegBootstrap function code snippet.

```java
1   public static void main(String[] args)  
2   {  
3       //...variable initialization and other functions...//
4       StartSOARegThread();
5   }
6   private static void StartSOARegThread(){
7       Thread expirychecker = new Thread(new RegExpiryChecker());
8       Thread locmsgreceiver = new Thread(new RegReceiver());
9       Thread locmsgdiscreceiver = new Thread (new RegDiscReceiver());
10      Thread regmsgsender = new Thread(new RegSender());
11      expirychecker.start();
12      regmsgsender.start();
13      locmsgdiscreceiver.start();
14      locmsgreceiver.start();
15   }
```

Referring to Listing 5.1, the RegBootstrap function invokes the StartRegSOAThread() method (line 4). The StartRegSOAThread() method is described in line 6-15. The StartRegSOAThread() method initiates the execution of four Java threads in the Reg SOA Manager, namely the RegExpiryChecker, RegReceiver, RegSender, and RegDiscReceiver.
SOA Implementation in SOSJ

The Global Service Repo, CD Macro-state Repo, and Subsystem Location Repo inside the GSR Data Structure are separate data structures implemented as three different Java class files, namely the SJServiceRegistry, RegCDStats, and RegSSAddr. On the other hand, the implementation of the AdvTimeStamp Repo is included in the SJServiceRegistry class. To be able to use JSON data structure in the global service registry application, the implementation uses the JSON for J2ME Java library [176]. The code snippet displaying some parts of the SJServiceRegistry class is shown in Listing 5.2.

Listing 5.2. The SJServiceRegistry class code snippet.

```java
import java.util.Enumeration;
import java.util.Hashtable;
import org.json.me."
import systemj.common.SOAFacility.Support.SOABuffer;

public class SJServiceRegistry {

    private static JSONObject currentServiceRegistry = new JSONObject();
    private static JSONObject advExpiryLength = new JSONObject();
    private static JSONObject advReceivedTime = new JSONObject();

    //...further class descriptions...//
}
```

Referring to Listing 5.2, the data structure used to store service descriptions and realize the Global Service Repo is the currentServiceRegistry of JSONObject type. The AdvTimeStamp Repo is realized through two data structures, namely the serviceExpiryLength and the advReceivedTime. The serviceExpiryLength stores each advertisement expiry time according to each subsystem associated to each advertisement. The advReceivedTime stores the timestamp when an advertisement is received.

The Mailbox in the global service registry application is implemented as a Java class named SOABuffer. The SOABuffer class has a number of data structures and get and set methods which implement the Mailbox function and provide the mechanisms to access its content. The code snippet describing some parts of the SOABuffer class is shown in Listing 5.3. Referring to Listing 5.3, the Mailbox uses the DiscMsgBuffer (line 4) to store Discovery messages received by the global service registry application. The RegNotify is a flag which indicates whether the RegSender transmits Notify (line 5). The Java Objects defined in line 7-8 are used as synchronizers to allow safe and synchronized access on the data structures defined in line 4-5. The get and set methods providing the mechanisms to access and modify the data structures in line 4-5 are described in line 11-35. The SetRegNotify() and GetRegNotify() methods (line 11-21) sets and obtains the RegNotify flag, respectively. The putDiscMsgToDiscBuffer() and getAllDiscMsgFromBuffer()
SOA Implementation in SOSJ

methods (line 23-35) stores a Discovery message to the DiscMsgBuffer and obtains all stored Discovery messages in the DiscMsgBuffer, respectively.

Listing 5.3. Parts of the SOABuffer class code snippet.

```java
public class SOABuffer {
    private static Vector DiscMsgBuffer = new Vector();
    private static boolean RegNotify = false;

    private static final Object RegNotifyLock = new Object();
    private static final Object DiscMsgBufferLock = new Object();
    //…further data structures variables…/

    public static void SetRegNotify (boolean stat){
        synchronized(RegNotifyLock){
            RegNotify = stat;
        }
    }

    public static boolean GetRegNotify(){
        synchronized(RegNotifyLock){
            return RegNotify;
        }
    }

    public static void putDiscMsgToDiscBuffer(JSONObject jsMsg){
        synchronized (DiscMsgBufferLock){
            DiscMsgBuffer.addElement(jsMsg);
        }
    }

    public static Vector getAllDiscMsgFromBuffer(){
        synchronized (DiscMsgBufferLock){
            Vector vec = DiscMsgBuffer;
            DiscMsgBuffer = new Vector();
            return vec;
        }
    }

    //…further class description…/
}
```

The SOA Message Generator of the global service registry is implemented as a Java class which provides the functionality to generate SOA messages, particularly Discovery Reply, Request for Advertisement, Beacon, and Notify. The code snippet describing some available methods in the SOA Message Generator class is shown in Listing 5.4. As shown in Listing 5.4, Request for Advertisement message is created using the ConstructReqAdvertisementMessage() method described in line 1-13, Notify message is created using the ConstructRegNotifyMessage() method described in line 15-25, Beacon message is created using the
SOA Implementation in SOSJ

ConstructRegistryBeaconMessage() method described in line 27-38, and Discovery Reply message is created using the ConstructDiscoveryReplyMessage() method described in line 40-63.

Listing 5.4. The code snippet of methods in the SOA Message Generator class.
The `RegDiscReceiver`, `RegReceiver`, `RegSender`, and `RegExpiryChecker` are implemented as four separate Java threads. The RegDiscReceiver implementation complies with the flowchart shown in Figure 5.2. When the RegDiscReceiver thread is initiated, it creates a UDP/IP socket which will be used to receive Discovery messages. If a Discovery message is received, the message is stored in the Mailbox to be processed by the RegSender thread. Then, the RegDiscReceiver thread goes back to wait for the next Discovery messages to be received. The code snippet describing the implementation of the RegDiscReceiver thread can be found in [177].

![Flowchart of RegDiscReceiver thread](image)

Figure 5.2. The process flowchart of the RegDiscReceiver thread.

The implementation of the RegReceiver thread complies with the process flowchart shown in Figure 5.3. Referring to Figure 5.3 on the left side, the RegReceiver thread begins by initiating a UDP/IP socket used to receive Advertisement message. The thread waits until it receives an Advertisement message. When an Advertisement message is received, the thread updates the data
structures in the GSR Data Structure (represented by the UpdateRepos process). The UpdateRepos process is further detailed on the right side of Figure 5.3. The UpdateRepos process consists of three consecutive updates on different data structures, the Global Service Repo (indicated by the UpdateServiceRepo), the AdvTimeStamp Repo (indicated by the UpdateAdvExpireRepo), and the CD Macro-state Repo and Subsystem Location Repo (indicated by the UpdateCDMSRepo & SSLocRepo). The code snippet describing the RegReceiver class can be found in [177].

![UpdateRepos flowchart](image_url)

Figure 5.3. The process flowchart of the RegReceiver class.

The implementation of the RegSender thread complies with the process flowchart shown in Figure 5.4. Referring to Figure 5.4, the RegSender thread starts by checking the Mailbox (in the data structure of SOABuffer class) for any Discovery messages received by the RegDiscReceiver thread. If one or more Discovery messages are present in the Mailbox, the thread will create Discovery Reply messages for each individual Discovery messages and transmit individual Discovery Reply messages towards the requesting parties. If no Discovery messages are present in the Mailbox or after Discovery Reply messages have been transmitted, the thread obtains the OS timestamp of recent Beacon transmission and compared with the current OS timestamp to check whether Beacon needs to be re-transmitted (based on the predetermined time period of Beacon transmission). If the OS time and the last Beacon transmission timestamp indicate that Beacon needs to be retransmitted, the Beacon message is created and broadcasted. Else, the thread accesses the Mailbox to check whether Notify needs to be transmitted. If Notify needs to be transmitted, Notify message is created and then transmitted (in this case broadcasted) by the RegSender thread. After the Notify message has been transmitted or if the transmission of Notify
message is not needed, the thread checks the timestamp of when individual Advertisements are received and their respective expiry time to determine whether there are any nearly expired Advertisements. If so, the thread creates Request for Advertisement message(s) and transmits them. After the Request for Advertisement messages are sent or if the transmission of the Request for Advertisement is not needed, the thread returns to first process. The implementation of the RegSender thread can be found in [177].

Figure 5.4. The RegSender thread process flowchart.
5.2 **SOA RTS**

The SOA RTS handles and implements the SOA functionalities in the SOSJ RTS. The SOA RTS communicates with the Dynamic RTS (part of the SOSJ RTS which handles dynamic behaviours) via Mailbox, as shown in Figure 5.5.

![Figure 5.5. Complete SOSJ RTS block diagram.](image)

The implementation of the Mailbox in the SOSJ RTS is similar with the global service registry. The Mailbox in the SOSJ RTS is used by different components in the SOSJ RTS to communicate and exchange information between each other and is implemented as two Java classes. These Java classes are namely SOABuffer, which is specifically used to access the SOA functionality handled by the SOA RTS, and CDLCBuffer, which is specifically used to access the dynamic behaviour handling functionality handled by the Dynamic RTS, which implementation will be described in Chapter 6. The code snippet describing the CDLCBuffer can be found in [177].

### 5.2.1 Design and Architecture

The architecture of the SOA RTS and the interactions between its components and other parts in the SOSJ RTS are shown in Figure 5.6. Referring to Figure 5.6, the **SOA RTS** consists of several parts, the **Internal Service Registry, SOA Function Call, SOA Message Generator, and SOA Manager**. The Internal Service Registry is the local service registry in each SOSJ RTS. The Internal Service Registry consists of four different data structures, the **Internal Service Repo**, **Internal CD Macro-state Repo**, **GSR Loc Repo**, and **Internal Subsystem Location Repo**. The Internal Service Repo stores the service description of services provided by CDs in the subsystem and of services provided by CDs belonging to other subsystems which will be invoked by CDs in
SOA Implementation in SOSJ

the subsystem. The Internal CD Macro-state Repo stores the information regarding the macro-state of CDs obtained from Discovery Reply. The GSR Loc Repo contains the information regarding the physical location (execution platform location) of the global service registry. The Internal Subsystem Location Repo stores the information regarding the physical location of other subsystems obtained from Discovery Reply. The **SOA Function Call** is a class which implements the programming constructs presented in Table 5.1 which allow CDs to use the features provided by the SOA RTS. The **SOA Manager** consists of three parts, the **SOA Receiver**, the **SOA Sender**, and **Discovery Handler**. The SOA Receiver is a thread that receives SOA messages depicted as SOAIn, which represents Beacon, Notify, and Request for Advertisement. The SOA Sender is a thread that transmits Advertisement to the global service registry. The Discovery Handler represents dedicated signal classes which implements the physical interfacing of the output and input signals used for the transmission of Discovery message and receiving Discovery Reply message. Note that the SOA Receiver and SOA Sender are initiated by the bootstrap function of the original SystemJ RTS. The description and implementation of the original SystemJ RTS is described in Appendix A.

![SOA RTS Architecture](image)

Figure 5.6. The architecture of SOA RTS.

### 5.2.2 Implementation

The Internal Service Repo, Internal CD Macro-state Repo, GSR Loc Repo, and Internal Subsystem Location Repo are implemented as four different Java classes namely the SJServiceRegistry, RegAllCDStats, SJRegistryEntry, and RegAllISSAddr. The SJServiceRegistry of the SOSJ RTS has the same implementation with the SJServiceRegistry of the SOSJ Global
Service Registry, while the implementation of the RegAllCDStats, SJRegistryEntry, and RegAllSSAddr can be found in [177].

The SOA Sender and SOA Receiver in the SOA Manager are implemented as two separate Java threads namely SOSJSOASender and SOSJSOAReceiver. The SJSOAReceiver thread follows the process flowchart shown in Figure 5.7. Referring to Figure 5.7, the SOSJSOAReceiver starts by creating a UDP/IP socket port to receive Request for Advertisement, Beacon, and Notify messages. Next, the thread waits to receive the aforementioned SOA messages. If a Notify message is received, the Notify message identifier is stored in the Mailbox. If a Beacon is received, the identifier and physical address of the global service registry is stored if there is no global service registry information is stored in the GSR Loc Repo, otherwise the message is ignored. If a Request for Advertisement is received, there are three possible process flows, due to the possibility of more than one subsystems deployed in a single machine. The following process flow may proceed to either of the three possible scenarios based on the destination subsystem name contained in the Request for Advertisement message and the physical address of the message origin. If the destination subsystem name matches with the subsystem’s name, the message is stored in the Mailbox. If the destination subsystem name contained in the Request for Advertisement message doesn’t match the subsystem’s name and the message is sent from a machine with different physical address than the subsystem’s, the message is retransmitted via multicast (Transmit Multicast). If the destination subsystem name contained in the Request for Advertisement message doesn’t match the subsystem’s name and the message comes from an origin with the same physical address with the subsystem (in case of retransmitted via multicast Request for Advertisement message packet), the message is ignored. The code snippet describing the implementation of the SOSJSOAReceiver class can be found in [177].

---

14 This is because in case of more than one subsystems in the same machine, there will be more than one SOSJSOAReceiver threads which listen to the same physical socket, hence the Request for Advertisement message may be received by either of the SOSJSOAReceiver threads of one of the subsystems. Since there is no way to ensure that the thread associated with the destination subsystem will always receive the Request for Advertisement in this case, if the Request for Advertisement message is received by the thread not associated to the destination subsystem, the message packet is retransmitted via multicast thus the SOSJSOAReceiver thread associated with the destination subsystem can receive.
Start
Create a UDP/IP Socket

Wait to Receive Message

If Request for Advertisement is Received
If Beacon is Received
If Notify is Received

If destination SS matches
If destination SS doesn’t match && the origin addr not equal with the destination

Store GSR Detail to Repo

If no GSR is present
If GSR is present

Store Notify Message ID

Transmit Multicast

Figure 5.7. The SOSJSOAReceiver thread process flowchart.

The SJSOASender thread follows the process flowchart shown in Figure 5.8. Referring to Figure 5.8, the thread starts by checking the Mailbox if any Request for Advertisement messages have been received and stored in the Mailbox. The thread also obtains the last timestamp of Advertisement transmission to check whether the transmission of Advertisement has to be performed and if there are requests to transmit Advertisement (which may come from within the SOSJ RTS or by CDs). If any of the aforementioned conditions is satisfied, the thread creates Advertisement message and transmits the Advertisement message. Otherwise, if neither applies, the thread goes back to the initial process.
The Discovery Handler is implemented as two different signal classes. One signal class, namely the TransmitDisc, handles the physical signal implementation of the SOSJ output signal used to transmit Discovery, while the other signal class, namely ReceiveDisc, handles the physical signal implementation of the SOSJ input signal used to receive Discovery Reply. The implementation is designed to have two distinct output and input signal names used for the transmission of Discovery and receiving Discovery Reply, namely “SOSJDisc” and “SOSJDiscReply”, respectively, in SOSJ, hence every output and input signals in SOSJ named SOSJDisc and SOSJDiscReply will be associated to the transmission of Discovery and receiving Discovery Reply. The code snippet describing the implementation of the TransmitDisc class is shown in Listing B.1. On the other hand, the code snippet describing the implementation of the ReceiveDisc class is shown in Listing C.1.

The SOA Function Call is implemented as a Java class named SOSJ containing a number of function calls for SOSJ CD to be able to perform service invocation (via signals or channels) and also use features offered by the SOA functionality. These function calls which are summarized in Table 5.1.
### Table 5.1. SOSJ SOA function calls

<table>
<thead>
<tr>
<th>Function Calls</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOSJ.ConfigureInvocChannel()</td>
<td>Make a request for channel reconfiguration to the RTS for service invocation via channel.</td>
</tr>
<tr>
<td>SOSJ.GetInvocChannelReconfigStat()</td>
<td>Obtains the status of channel reconfiguration process (successful or not) from the RTS.</td>
</tr>
<tr>
<td>SOSJ.CreateChanInvReqMsg()</td>
<td>Generate service invocation message for service invocation via channel.</td>
</tr>
<tr>
<td>SOSJ.CreateSigInvReqMsg()</td>
<td>Generate service invocation message for service invocation via signal.</td>
</tr>
<tr>
<td>SOSJ.CreateSigInvRespMsg()</td>
<td>Generate a response to service invocation message via signal.</td>
</tr>
<tr>
<td>SOSJ.StoreService()</td>
<td>Stores service description into the internal service registry in the RTS.</td>
</tr>
<tr>
<td>SOSJ.CreateChanInvRespMsg()</td>
<td>Generate a response to service invocation message via channel.</td>
</tr>
<tr>
<td>SOSJ.GetAction()</td>
<td>Gets the ‘action’ name included in service invocation message</td>
</tr>
<tr>
<td>SOSJ.GetData()</td>
<td>Gets any data/value included in service invocation message</td>
</tr>
<tr>
<td>SOSJ.SetCDServVisib()</td>
<td>To include/exclude service description of services of a CD for Advertisement</td>
</tr>
<tr>
<td>SOSJ.TriggerRefreshAdv()</td>
<td>To trigger the transmission of Advertisement.</td>
</tr>
<tr>
<td>SOSJ.GetNotif()</td>
<td>Used by CD to check if Notify has been received</td>
</tr>
</tbody>
</table>

#### A) Service invocation via signals message structure

Service invocation (via signal) request message is generated using the SOSJ.CreateSigInvReqMsg() function call. The structure of the generated message is shown in Listing 5.5.

Listing 5.5. The structure of service invocation (via signal) request message.

```
1  {  
2      "action" : "CloseDiverter",  
3      "msgID" : "7481367832"  
4  }  
5  
6  {  
7      "action" : "ActuateConveyor",  
8      "msgID" : "7484783926",  
9      "data" : "10"  
10 }  
11  
12  {  
13      "action" : "ActuateConveyor",  
14      "msgID" : "7484783926",  
15      "data" : "10",  
16      "respAddr", "192.168.1.80",  
17      "respPort", "7896",  
18    }
```

104
SOA Implementation in SOSJ

Referring to Listing 5.5, the service invocation (between consumer and provider in the same CD via signal) request message structure complies with either the example in line 1-4 or line 6-10. The ‘action’ key defines the action name which will be invoked to obtain results. The ‘msgID’ key defines the message identifier. The ‘data’ key defines particular data/value to be passed to the provider side. On the other hand, the service invocation (between consumer and provider in different SOSJ programs via signal) request message structure complies with the example in line 12-20. The message contains the following JSON keys. The ‘action’ key (line 13) defines the action name which will be invoked to obtain results or perform functions. The ‘msgID’ key (line 14) defines the message identifier. The ‘data’ key (line 15) refers to the data being passed to the reaction provider side. If no data is included by the consumer CD, then the ‘data’ key is not present in the message. The ‘respAddr’ key (line 16) refers to the machine address (physical location) where the consumer CD runs. The ‘respPort’ key (line 17) refers to the physical port number used by the consumer CD to receive reply. The ‘destAddr’ key (line 18) refers to the machine address (physical location) where the provider CD runs. Finally, the ‘destPort’ key (line 19) refers to the physical port number used by the provider CD to receive invocation request message. The example described in line 1-4 shows an example of service invocation request message for closing the diverter D1 which doesn’t require any values/data to be passed to the provider to obtain results. The example described in line 6-10 presents an example of service invocation request message to actuate conveyor (e.g. CB1 which the action requires a value/data, in this case the conveyor’s rotational speed) for invocation within the same CD, while the example described in line 12-20 shows an example of service invocation request message for invocation involving consumer and provider in different SOSJ programs.

Once a service invocation request message is received on the provider’s side, the provider’s side generates service invocation response message (using the SOSJ.CreateSigInvRespMsg()) to be transmitted towards the consumer’s side. The structure of the service invocation response message is shown in Listing 5.6.

```
1  { 
2      "status" : "ACK", 
3      "msgID" : "7481367832"
4  } 
5
```

Listing 5.6. The structure of service invocation (via signal) response message.
SOA Implementation in SOSJ

Referring to Listing 5.6, the structure of service invocation (between consumer and provider in the same CD via signal) response message may comply with either the example described in line 1-4 or line 6-10. The ‘status’ key defines the status of the service invocation process. The status returns ‘ACK’ if the invocation is completed. The ‘msgID’ key contains the same message identifier which corresponds to the service invocation request message. The ‘data’ key defines any values/data returned by the provider’s side (if any). On the other hand, the service invocation (between consumer and provider in different SOSJ programs via signal) response message structure complies with the example in line 12-17. The ‘destAddr’ key (line 15) refers to the machine address (physical location) where the consumer CD runs. Finally, the ‘destPort’ key (line 16) refers to the physical port number used by the consumer CD to receive invocation response message.

B) Service invocation via channels message structure

Service invocation (via channel) request message is generated using the SOSJ.CreateChanInvReqMsg() method. The structure of the generated message is shown in Listing 5.7.

Listing 5.7. The structure of service invocation (via channel) request message.
Referring to Listing 5.7, the service invocation (via channel) request message structure may have the structure of either the one shown in line 1-7 or line 9-16. The ‘action’, ‘msgID’, and ‘data’ keys of the service invocation (via channel) request message have the same purposes with the ones used for service invocation (via signal) request message. The ‘respSSName’ key defines the subsystem name of the consumer side. The ‘respCDName’ defines the CD name of the consumer side. The ‘respChanName’ key defines the name of the input channel used to receive results/response from the provider side.

Similar to service invocation via signal, the provider’s side generates service invocation response message to be transmitted towards the consumer’s side after a service invocation request message is received on the provider’s side using the SOSJ.CreateChanInvRespMsg() method. The structure of service invocation (via channel) response message is shown in Listing 5.8.

Listing 5.8. The structure of service invocation (via channel) response message.

```json
1  {
2   "status" : "ACK",
3   "msgID" : "7481367812",
4  }
5
6  {
7    "data" : "high",
8    "msgID" : "7481367733",
9    "status" : "ACK",
10  }
```

Referring to Listing 5.8, the service invocation (via channel) response message has the same structure with the service invocation (via signal) response message, containing the status, msgID, and data keys (if any), sent from the provider side to the consumer side. Service invocation via channel requires the reconfiguration of output and input channels used by both communicating provider and consumer. The functionality which performs the channel reconfiguration process is part of the Dynamic RTS and will be elaborated in detail in Section 6.2.2.

5.2.3 Describing CD in SOSJ and Utilizing the SOA Function Calls

As a guideline in describing CD with service consumer role, the CD needs to use two signals, namely SOSJDisc and SOSJDiscReply, which are dedicated SOSJ signals used for the purpose of transmitting Discovery and receiving Discovery Reply, respectively. On the other hand, as a guideline to describe CD which invokes service offered by different CD via channel, the CD needs
SOA Implementation in SOSJ

to use a pair of channels, input and output. The output channel is used to send service invocation message, while the input channel is used to receive reply from the provider CD. On the other hand, to describe CD with service provider which offers services invokable by different CD, a pair of channels, input and output, needs to be used by the CD. The input channel is used to receive service invocation message, while the output channel is used to send reply. Figure 5.9 shows an illustration showing the CD utilizing signals and channels as described earlier. On the left, the consumer CD named ‘Cons’ uses InvSend channel to transmit service invocation message and InvRespRec channel to receive reply. On the right, the provider CD named ‘Prov’ uses InvRec channel to receive service invocation message and InvRespSend to send reply. In case of service invocation between CDs in different SOSJ programs, the InvSend and InvRespRec represent the output and input signals used by the Cons CD to transmit service invocation request message and receive service invocation response message, while the InvRec and InvRespRec represent the input and output signals used by the Prov CD to receive service invocation request message and transmit service invocation response message, respectively.

![Graphical illustration of the ‘template’ of consumer and provider CD.](image)

To showcase how the SOA functionality and function calls in SOSJ are used. Listing 5.9, Listing 5.10, and Listing 5.11 present an example of how CD can find service and perform service invocation using the provided SOA functionality and function calls.

Listing 5.9. Code snippet showing an example on how to perform service invocation.

```plaintext
1 CD1(
2     input String signal SOSJDiscReply;
3     output signal SOSJDisc;
4     output String channel Invoke1ReqCh;
5     input String channel Invoke1RespCh;
6     output String signal ActuateConveyor;
7 )->{
8     signal ReqInvoke1S;
9     signal RespInvoke1S;
10    }
```

108
SOA Implementation in SOSJ

{ //...variable instantiation...//
  emit SOSJDisc;
  await (SOSJDiscReply);
  String serv = (String)#SOSJDiscovery;
  Hashtable matchedRes1 = DoMatching(serv,"PE",BLoaderALocation);
  //...get service matching result and further behaviour description...://
  boolean reconfstat1 = SOSJ.ConfigureInvocChannel("CD1","Invoke1ReqCh",servCDName,servChanName);
  if(reconfstat1){
    pause;
  }
  boolean reconfstat2 = SOSJ.GetInvocChannelReconfigStat(servCDName,servChanName);
  if(reconfstat2){
    String servSSName = matchedRes1.get("servSSName").toString();
    String ChReqMsg = SOSJ.CreateChanInvReqMsg("CD1","Invoke1RespCh",actNamePE);
    send Invoke1ReqCh(ChReqMsg);
    receive Invoke1RespCh;
    //...further behaviour description...://
   Hashtable matchedRes2 = DoMatching(serv,"conveyor",ConveyorLocation);
    String actNameCnv = matchedRes2.get("actionName").toString();
    String SReqMsg = SOSJ.CreateSigInvReqMsg(actNameCnv ,"10");
    emit ReqInvoke1S(SReqMsg);
  }
  SOSJ.StoreService(servDescPE);
  SOSJ.StoreService(servDescCnv);
}
//...further behaviour description...//
pause;
}
||
{ //...further variable instantiation and behaviour description...://
  while(true){
    await (ReqInvoke1S);
    String recvdReqMsg = (String)#ReqInvoke1S;
    String actName=SOSJ.GetAction(recvdReqMsg);
    if(actName.equals("ActuateForward")){
      String val = SOSJ.GetActionData(recvdReqMsg);
      String respMsg = SOSJ.CreateSigInvRespMsg(recvdReqMsg);
      emit ActuateConveyor(val);
      emit RespInvoke1S(respMsg);
    }
    pause;
  }
}
CD2(|
  input String signal SOSJDiscReply;
  output signal SOSJDisc;
  output String signal Invoke2ReqS;
  input String signal Invoke2RespS;
})->{
  while(true){
    //...further variable instantiation and behaviour description...://
SOA Implementation in SOSJ

```java
68       String SReqMsg = SOSJ.CreateSigInvocReqMsg("CD2","Invoke2RespS",dAddr, dPort,actName,"25");
69       emit Invoke2ReqS(SReqMsg);
70       await (Invoke2RespS);
71          //...further behaviour description.../
72          pause;
73     } }
74 }
```

The code snippets show how consumers are made aware of available services and dynamically invoke services via channel and signal, in this case, to actuate conveyor CB1 once a bottle is detected by PE1 sensor in front of the Bottle Loader in point A (referring to the motivating scenario illustrated in Figure 1.1), and actuate conveyor CB2. The behaviour which provides service to actuate conveyor CB1 is described in line 45-58 of Listing 5.9, the behaviour which provides service to detect the presence of bottle on conveyor CB1 using PE1 is described in line 1-19 of Listing 5.10, and the behaviour (CB2CD) which provides service to actuate conveyor CB2 is described in Listing 5.11. The behaviour which invokes the service to detect bottle presence and actuate the conveyor CB1 is described in CD1 in line 10-43 of Listing 5.9, while the behaviour which invokes the service to actuate the conveyor CB2 is described in CD2 in Listing 5.11.

Listing 5.10. Code snippet on service invocation via channel – PE1CD.

```java
1 PE1CD(
2       input String signal PE1Sensor;
3       input String channel RecInvPE1Ch;
4       output String channel RespInvPE1Ch;
5 )->{
6     {
7         while(true){
8            receive RecInvPE1Ch;
9            String recvdMsg = (String)#RecInvPE1Ch;
10           await (PE1Sensor);
11           String pe1val = (String)#PE1Sensor;
12           SOSJ.ConfigureInvocChannel("PE1CD","RespInvPE1Ch",recvdMsg);
13           pause;
14           String respMsg = SOSJ.CreateChanInvRespMsg(recvdMsg,pe1val);
15           send RespInvPE1Ch(respMsg);
16           pause;
17          }
18      }
19  }
```

Listing 5.11. Code snippet on service invocation via signal (between different SOSJ programs) – CB2CD.

```java
1 CB2CD(
2       input String signal RecInv2S;
3       output String signal RespInv2S;
4       output String signal ActConv2;
```
To achieve the scenario of actuating CB1 once a bottle is detected by PE1 sensor, the CD1 checks the identifier of the Notify message last received by the RTS to see if new Notify message has been received. If a new Notify message is received, CD1 attempts to obtain the list of services advertised to the global service registry using the SOSJDisc and SOSJDiscReply signals (line 13-14 of Listing 5.9). Note that the SOSJDisc and SOSJDiscReply are dedicated signals used for the purpose of transmitting Discovery and receiving Discovery Reply, respectively. Once Discovery Reply is received and the list of advertised services is obtained (line 15 of Listing 5.9), CD1 performs service matching (line 16 of Listing 5.9) to find the PE1 sensor service using a Java method ‘DoMatching’ (which implementation is programmer-dependent and, in this case, implemented by the programmer). For example, the method takes into account the physical sensor type and the position of the Bottle Loader station currently in use. Once the service matching returns the result, the consumer behaviour proceeds to service invocation process, i.e., invoke the PE1 sensor service. In this example, invoking the PE1 sensor service is performed via channel, since the PE1 service behaviour is described in another CD named ‘PE1CD’ which is described in Listing 5.10. The RTS requires a number of parameters for the channel reconfiguration of the output and input channel ports used by the consumer and the PE1 provider to initiate. The RTS requires the name of the CD that invokes the service and the channel name used to invoke the service. Both are obtained from the service matching results and defined in the variable ‘servCDName’ and ‘servChanName’. Next, the consumer makes a request for channel reconfiguration using the function call provided by the RTS (line 18 of Listing 5.9). If the function call returns true (meaning that the request is passed to the RTS for channel reconfiguration process), the CD checks whether the channel reconfiguration process is successful using a function call (line 21 of Listing 5.9). If the channel reconfiguration process is successful, the consumer generates a service invocation message using a function call (line 23-24 of Listing 5.9), with the parameters of the CD name where the consumer behaviour resides and the channel name that
accepts result sent from the provider, and the action name. Then the consumer sends the service invocation message (line 25 of Listing 5.9). When the consumer behaviour receives response from the provider (line 26 of Listing 5.9), it can proceed to obtain the sensing result. If the result depicts that a bottle is detected by PE1 sensor, the consumer performs another service matching to find the conveyor service (line 28-29 of Listing 5.9). Then, the consumer generates and sends a service invocation request message to actuate conveyor CB1 with action name represented by the ‘actNameCnv’ variable and with speed of ‘10’ (line 30 of Listing 5.9). The consumer may store the service description of matching services using the function calls shown in line 34-35 of Listing 5.9.

To actuate conveyor CB2 through invocation via signal (between consumer and provider in different SOSJ programs), CD2 generates the invocation request message (line 68 of Listing 5.9) after it performs service discovery and matching to find the CB2 service and its service interface details (e.g. its physical address represented as ‘dAddr’ and physical port number represented as ‘dPort’ which are found through service matching) through the similar mechanisms shown in line 13-16 of Listing 5.9. Then, the CD transmits the invocation message via signal Invoke2ReqS (line 69 of Listing 5.9) and waits to receive response via signal Invoke2RespS (line 70 of Listing 5.9). The CB2CD (which provides the CB2 service) receives invocation request message via signal RecInv2S (line 8 of Listing 5.11) and transmits invocation response message via signal RespInv2S (line 13 of Listing 5.11).

The way to dynamically discover and invoke service in Listing 5.9 is also amenable when a provider fails and the consumer has to find another service offering similar functions from different providers.

**5.3 Concluding Remarks**

This chapter has described parts of the implementation of the SOSJ framework, which achieves the SOA functionalities described in Chapter 3. They include the SOSJ Global Service Registry, which consists of data structures that stores service description of advertised services and functionalities to realize the SOA-based interactions with service consumer and service provider side, and the SOA RTS which handles and implements SOA functionalities in the SOSJ RTS. The rest of the implementation of SOSJ framework which provide the mechanisms to handle dynamic behaviours will be described in the next chapter (Chapter 6).
6 Dynamic Behaviour Handling in SOSJ

The SOSJ programming paradigm is created from the integration of the SOA paradigm and SystemJ. Since the SOA paradigm doesn’t define any particular solutions/mechanisms in dealing with dynamic behaviours, additional provisions, i.e. CD macro-states and their transitions, are proposed and described in Chapter 4 which extend the semantics of CD in SystemJ to cater different scenarios of dynamic behaviour. However, the functionalities which handle and achieve dynamic behaviours in SOSJ are not described in the chapter. This chapter describes these functionalities which are developed based on the proposed CD macro-states and their transitions. These functionalities are introduced as further extension of the SOSJ RTS and referred as the Dynamic RTS. The Dynamic RTS consist of a number of Java threads and data structures to handle link creation, dynamic creation, suspension, resumption, termination, and migration of CD and also channel reconfiguration process. The Dynamic RTS also provides a number of function calls which can be invoked to trigger dynamic behaviours.

This chapter is broken down into two sections. Section 6.1 describes the design and architecture of the Dynamic RTS, while Section 6.2 details the implementation of the Dynamic RTS, function calls to trigger dynamic behaviours, and describes how they are used in an example.

6.1 Dynamic RTS Design and Architecture

The architecture of the Dynamic RTS and the relationships between its components and other parts of the SOSJ RTS is shown in Figure 6.1 (taken from [178]). Referring to Figure 6.1, the Dynamic RTS consists of a number of threads and functionalities that facilitate dynamic behaviours. Components in the Dynamic RTS interact with the SOA RTS through the Mailbox, however, the Reconfiguration Manager in the Dynamic RTS accesses the internal service registry (which is a part of the SOA RTS) directly without going through the Mailbox. The Reconfig Req Receiver is a thread acting as a server which receives incoming request for performing dynamic behaviours from an external source (e.g. human operator). The Dynamic
Function Call provides new programming constructs that is available to programmers to support dynamic behaviours and trigger reconfiguration. The Dynamic RTS XML Parser provides the mechanism to parse and obtain information from CD and subsystem configuration and service description (which are written in the XML format).

Figure 6.1. The Dynamic RTS as part of the SOSJ RTS (in red colour).

The SOSJ Config comprises three different data structures, the CD Config Repository, the CD Descriptor Storage, and the Link Config Repository. The CD Descriptor Storage is a data structure that stores CD descriptors, Java objects which contain data and execution state of CDs in the subsystem and also their signal and channel communication status. This data structure is particularly important as it enables suspension and resumption of CD execution while retaining all data and execution state, and also strong migration of CD. The Link Config Repository is a data structure which stores the configuration of links used by the subsystem. The CD Config Repository is a data structure which stores the configuration of all CDs in the corresponding subsystem, which includes the mapping of their signals and channels.

The Reconfiguration Manager is the main component in charge of executing all necessary functionalities to ensure correct reconfiguration. It is responsible for including CD for or excluding CD from execution/scheduling based on its macro-states, storing and fetching CD descriptors to and from the CD Descriptor Storage, altering the service description (when necessary) stored in
the internal service registry inside the SOA RTS, and configuring and establishing signals, channels, and links when necessary following the configuration stored in the CD Config Repository.

The Migration Manager consists of two functionalities which handle the migration between the source and destination subsystems, the Migration Receiver and the Migration Sender. The Link & Migration Request Handler receives any request message to initiate migration (Mig Req Msg) and link creation (Link Req Msg). It transmits a reply message (Mig Resp Msg) as a response to the request for migration process. The Link Negotiation Manager consists of three functionalities. The Link Requester initiates the handshake for link creation between two subsystems by transmitting a request message (Link Req Msg), which will be received by the Link & Migration Request Handler of the partner subsystem, and receives reply from it (Link Resp Msg). When a reply is received, the Link Requester starts the Link Negotiator Sender thread, while the partner subsystem, after sending the reply to the requesting side, starts the Link Negotiator Receiver thread. Then, both sides perform the handshake process to create a mutual physical interface for a link to be used by both subsystems.

6.2 Dynamic RTS Implementation

Due to the scale of Dynamic RTS’ implementation, the description of the implementation is presented according to each dynamic behaviours, i.e. CD creation, CD suspension, CD resumption, CD termination, and CD migration, and also related processes such as channel reconfiguration and link creation processes. The following function calls are introduced to SOSJ which can be used by CD to trigger dynamic behaviours. They are described in Table 6.1.

Table 6.1. SOSJ new function calls to trigger dynamic behaviours.

<table>
<thead>
<tr>
<th>Function Calls</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJ.CreateCD(CDName,CDMap,CDSD)</td>
<td>To create a CD with name CDName, the XML file name describing the CD configuration CDMap, and the XML filename describing the service description CDSD.</td>
</tr>
<tr>
<td>SJ.KillCD(CDName)</td>
<td>To remove a CD of name CDName</td>
</tr>
<tr>
<td>SJ.MigrateCD(CDName, CDMap, CDSD, Dest, MigType)</td>
<td>To migrate a CD of name CDName, with CD configuration described in the XML file name CDMap, service description described in the XML file name CDSD, to another subsystem defined in Dest, and the migration type defined in MigType.</td>
</tr>
<tr>
<td>SJ.SuspendCD(CDName)</td>
<td>To temporarily suspend a CD execution with name CDName</td>
</tr>
</tbody>
</table>
Results and Discussions

| SJ.WakeUp(CDName) | To resume execution of a CD with name CDName |

The Dynamic RTS XML Parser is implemented as a new Java class named CDLCMapParser, which contains the functionality to obtain parameters defined in the CD configuration and service description written in XML format, while the CD and subsystem configuration is stored in a data structure class named SJSSCDSignalChannelMap. The implementation of the both CDLCMapParser and SJSSCDSignalChannelMap can be found in [177]. For the Dynamic RTS implementation to make sense, it is necessary to refer to the original SystemJ RTS implementation, which can be found in Appendix A.

6.2.1 Link Creation

Link creation process is run when no link is available to physically exchange data sent to or received by CDs belonging to different subsystems through channels. Link creation process involves the Link Config Repository, Link Requester, Link & Migration Request Handler, Link Negotiator Sender, and Link Negotiator Receiver.

The Link Requester is implemented as a Java class named LinkCreationSender, which contains functionality to transmit request to initiate handshake for link creation and also receives response to the request to initiate link creation process. The code snippet describing the LinkCreationSender can be found in [177]. The LinkCreationSender class sends a request to create link to the Link & Migration Request Handler, which is implemented as a Java thread named MigrationAndLinkReqMsgThread. The code snippet describing the part of the MigrationAndLinkReqMsgThread class that communicates with the LinkCreationSender in realizing the link creation process can be found in [177]. Once a request to perform link creation is received and acknowledged, the MigrationAndLinkReqMsgThread initiates a Java thread called the LinkCreationReceiverThread. The thread handles the handshake function of the responding party to establish the new link for the requesting and responding party to agree on mutual physical communication interfaces to be used for link. The code snippet describing the LinkCreationReceiverThread can be found in [177]. The link creation process follows the sequence diagram shown in Figure 6.2.
Results and Discussion

Referring to Figure 6.2, the LinkCreationSender finds a UDP/IP physical port which will be used to receive response message from the MigrationAndLinkReqMsgThread (getAvailablePort()). Then, the LinkCreationSender finds a TCP/IP physical port to be used to receive data from the responding party (partner) during link creation handshake process (getAvailableReceivingPort()). Next, the CreateLinkReqMsg() is invoked to generate a request to initiate link creation. Once the request to initiate link creation message is created, the LinkCreationSender invokes the TransceiveIsLinkCreationFree() method. The method transmits the request to initiate link creation message and then waits to receive reply. The ReceiveMigOrLinkMsg() method of the MigrationAndLinkReqMsgThread receives the request to initiate link creation message. The request message is then passed to the ResponseMigOrLinkReq() method. The ResponseMigOrLinkReq() method generates a response message which is then transmitted to the requesting side. If both the requesting and responding parties agree to initiate link creation process (indicated by ‘If Partner SS Link Creation Function Not Used’ in Figure 6.2), the LinkCreationSender invokes the ExecuteLinkCreation() method which handles the link creation handshake process of the requesting party. Meanwhile, the MigrationAndLinkReqMsgThread starts the LinkCreationReceiverThread which handles the link creation handshake process of the responding party after the TCP/IP physical socket used to receive data from the requesting side during the link creation handshake is established. The link creation handshake sequence diagram is shown in Figure 6.3.
If no physical port to receive in the SS

Start:
- Partner Subsystem Name
- Partner's Link TCP Port Alloc
- Subsystem Name
- Subsystem Link TCP Port Alloc

Check Port()

FindAvailablePort()
InstantiateTCPPortToReceive()
UpdateSSLocReg()
UpdateLinkReg()
SendPortNum()

RegAllSSAddr
TCPIPLinkRegistry

InstantiateTCPPortToSend()
UpdateSSLocReg()
UpdateLinkReg()
Results and Discussion

Figure 6.3. Sequence diagram of link creation handshake.

(b)
Results and Discussion

Referring to Figure 6.3, the link creation handshake begins with the transmission of a START message from the ExecuteLinkCreation(), which will be received by the LinkCreationReceiverThread. Then, both parties exchange the names of the subsystem and the list of links allocated in their partner subsystem’s RTS. Next, both parties check the list of their corresponding partner’s allocated links to determine whether their partner have links available to receive channel data (indicated by CheckPort() in Figure 6.3). Based on whether any links are present on both subsystems to communicate, there are several possibilities on how the handshake process proceeds as shown in Figure 6.3. If no physical TCP/IP socket is present (as link) for the corresponding subsystem to receive data from the partner subsystem (indicated by the “If no physical port to receive in the SS” in part (a) of Figure 6.3), the ExecuteLinkCreation() creates a new TCP/IP socket using an unbound TCP/IP physical port to be used as a new link (indicated by the FindAvailPort() method) to receive channel data from the partner subsystem’s side (indicated by the InstantiateTCPPortToReceive() method). Next, the partner subsystem’s physical address is stored in the RegAllSSAddr (indicated by the UpdateSSLocReg()) and the TCP/IP port number of the link is stored in the TCPIPLinkRegistry (indicated by the UpdateLinkReg() method). Finally the port number is sent to the LinkCreationReceiverThread (indicated by the SendPortNum()) for the RTS of the partner subsystem to use as link to transmit channel data to the corresponding subsystem (indicated by the InstantiateTCPPortToSend()), before storing the corresponding subsystem’s physical address and the port number (indicated by the UpdateSSLocReg() and UpdateLinkReg()).

If the corresponding subsystem has a TCP/IP socket allocated as link to receive channel data (indicated by “if physical port to receive in the SS exists” in part (b) of Figure 6.3) while the partner subsystem doesn’t have the TCP/IP port number to send channel data to the corresponding subsystem (indicated by “if no physical port in the partner SS to send data”), the ExecuteLinkCreation() method sends the physical port number used by the corresponding subsystem to receive channel data to the LinkCreationReceiverThread of the partner subsystem’s RTS for it to instantiate a TCP/IP socket to use to transmit channel data to the corresponding subsystem. Then, the corresponding subsystem’s physical address is stored in the RegAllSSAddr (indicated by the UpdateSSLocReg()) and the TCP/IP port number is stored in the TCPIPLinkRegistry (indicated by the UpdateLinkReg() method).

If the partner subsystem has no link available to receive channel data, the LinkCreationReceiverThread of the partner subsystem’s RTS triggers the creation of a TCP/IP socket using an unbound TCP/IP physical port (indicated by FindAvailPort()) to receive channel data from the corresponding subsystem (indicated by the InstantiateTCPPortToReceive()). Then,
Results and Discussion

the LinkCreationReceiverThread stores the partner subsystem’s physical address in the RegAllSSAddr (indicated by the UpdateSSLocReg()) and the TCP/IP port number of the link in the TCPIPLinkRegistry (indicated by the UpdateLinkReg() method) and sends the port number to the corresponding subsystem (indicated by the SendPortNum()) to use it for a TCP/IP socket to transmit channel data to the partner subsystem (indicated by the InstantiateTCPPortToSend()). Then, the ExecuteLinkCreation() method stores the partner subsystem’s physical address and the port number (indicated by the UpdateSSLocReg() and UpdateLinkReg()).

If the partner subsystem has a TCP/IP socket already allocated as a link to receive channel data (indicated by “if physical port to receive in the partner SS exists”), while the corresponding subsystem hasn’t allocated a link to send channel data to the partner subsystem (indicated by “if no physical port in the SS to send data”), the LinkCreationReceiverThread sends the port number to the corresponding subsystem’s side for the RTS of the corresponding subsystem to create a new link based on a TCP/IP socket of the same port number to transmit channel data to the partner subsystem. Then, the partner subsystem’s physical address is stored in the RegAllSSAddr (indicated by the UpdateSSLocReg()) and the TCP/IP port number of the link is stored in the TCPIPLinkRegistry (indicated by the UpdateLinkReg() method).

6.2.2 Channel Reconfiguration for Service Invocation

Apart from CD creation, suspension, resumption, termination, and migration, channel reconfiguration is also handled by the Dynamic RTS. Service invocation occurring between consumer and provider associated to different CDs involves channel reconfiguration. Channel reconfiguration for service invocation via channel follows the sequence diagram shown in Figure 6.4. Referring to Figure 6.4, when the ConfigureInvocChannel() is invoked during the execution time, it accesses the data structure in the CDLCBuffer class and stores the request to reconfigure one channel pair (an output channel of the consumer CD and an input channel of the provider CD) which will be used to transfer service invocation message from the consumer to the provider. The request is buffered to the data structure of the CDLCBuffer only if there is no existing request to reconfigure the same input channel of the provider CD, which the method returns a boolean with the value of true. Otherwise, the method discards the request and returns a boolean with the value of false. Based on the value returned by the method, the consumer CD is given the liberty to decide whether it will proceed with the service invocation (e.g. reattempt making the request to reconfigure the same channel in the next tick, or attempt another Discovery to find another
Results and Discussions

service). The code snippet describing the implementation that follows the sequence diagram detailing what happen during the reconfiguration time\textsuperscript{15} in Figure 6.4 is shown in Listing 6.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequence_diagram_channel_configuration.png}
\caption{Sequence diagram of channel configuration for service invocation via channel.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{code_snippet_channel_reconfiguration.png}
\caption{Code snippet describing the implementation of channel reconfiguration for service invocation.}
\end{figure}

\begin{verbatim}
1   public class SystemJProgram {
2     //...further class descriptions.../
3     if(!CDLCBuffer.IsInvokeServChanReconfigBufferEmpty()){
4         Hashtable vec = CDLCBuffer.GetReconfigInvokeServChanBuffer();
5         Scheduler scMod = getScheduler();
6         while(IMBuffer.getIMUpdatingFlag()){}
7         InterfaceManager imMod = IMBuffer.getInterfaceManagerConfig();
8         try {
9             Enumeration keysChanInvRec = vec.keys();
10            String locSSName = SJSSCDSignalChannelMap.getLocalSSName();
11            JSONObject jsAllMap = SJSSCDSignalChannelMap.getCurrentSignalChannelMapping();
12            JSONObject LocCDConfMap = jsAllMap.getJSONObject(locSSName);
13            while (keysChanInvRec.hasMoreElements()){
14                boolean isLocal = LocCDConfMap.has(PartnerCDName);
15            }\textsuperscript{15}
\end{verbatim}

\textsuperscript{15} See Figure 3.4
In case the buffer storing the request for channel reconfiguration is not empty, during the reconfiguration time, the functionality accesses the buffer to obtain all stored requests to reconfigure channel for service invocation (using the GetReconfigInvokeServChanBuffer() method, as shown in line 4 of Listing 6.1) after it checks the content of the buffer (using the IsInvokeServChanReconfigBufferEmpty(), as shown in line 3). Next, after the Scheduler and InterfaceManager instances are obtained (indicated by the GetScheduler() and getInterfaceManagerConfig() in Figure 6.4, implementation described in line 5-7 of Listing 6.1), the CD configuration is obtained (indicated by GetConfig(), implementation described in line 10-12 of Listing 6.1). Then, the functionality obtains each request fetched from the buffer and gets the configuration parameters of the output channel and input channel which will be involved in the service invocation stored together alongside the request in the buffer. Then, the functionality checks whether the provider CD belongs to the same subsystem with the consumer CD (line 15), and then passes the information to the ReconfigInvServChannel() method of the ClockDomainLifeCycleSigChanImpl class when it is invoked in line 16. The ReconfigInvServChannel() method updates and returns both the Scheduler and InterfaceManager instances to include the changes on the channels and also new links (if any) (line 17-18). Then, the Scheduler instance is set to be used for the upcoming execution time and the InterfaceManager instance is stored (indicated by updateScheduler() and SaveInterfaceManagerConfig() in line 19-20). Finally, the buffer containing the request to reconfigure channel is cleared (line 22).
Results and Discussion

If New Partner CD is Local

If New Partner CD is Active
getPartnerCDFromSchedObj()

If New Partner CD is Suspended
getCDDesc()

If InChan "occupied"
SetInChanOccupied()
AddInvokeServStatChanReconfig()

If InChan "not occupied"

If new partner is different than the old partner
getInChanObj()
getPartnerCDFromSchedObj()

If new partner is the same with the old partner

If Current Partner CD is Local
UnbindCurrentPartner()
updateCurrentPartnerConfig()

If Current Partner CD is not Local
TransmitUnbindPartnerCommand()
Results and Discussions

The implementation of the ReconfigInvServChannel() method follows the sequence diagram shown in Figure 6.5. Referring to Figure 6.5, the ReconfigInvServChannel() method starts by accessing the CD descriptor of the consumer CD and the output channel object. Then, after the name of the partner input channel and the partner CD that uses the input channel are found, the method checks whether the partner CD belongs to the same subsystem with the consumer CD. If the partner CD belongs to the same subsystem (in part (a) of Figure 6.5), the method fetches the

Figure 6.5. The ReconfigInvServChannel() sequence diagram.
CD descriptor of the provider CD and the input channel. Next, the method checks whether the input channel is in the occupied status. If the input channel is in the occupied status, the method stores a response to the CDLCBuffer through the AddInvStatChanReconfig() method (for the response to be accessible by the consumer CD using the GetInvocChannelReconfigStat(), which is also shown in the example in Listing 5.9) notifying that the input channel of the provider CD is in the occupied status. Otherwise, the method continues by checking whether the output channel is going to be bound with a different partner input channel.

If the new partner input channel has the same name with the current partner input channel, the method doesn’t change the output channel binding and sets the input channel to be in the occupied status. Otherwise, the method goes to check whether the current partner input channel is associated with the same subsystem with the output channel.

If the current partner input channel is associated to the same subsystem, the method is able to access the current partner input channel, unbind the current partner input channel from the output channel, and modifies the channel configuration of the partner input channel accordingly to indicate that the partner input channel has been unbound from the output channel. Otherwise, if the current partner input channel is associated to another subsystem, the method queues a command to be transmitted to the current partner input channel (via link) to unbind the current partner input channel from the output channel. Once the current partner input channel is unbind from or given a command to unbind from the output channel, the method accesses the input channel to be paired with the output channel. Then, both the output channel and the input channel are reconfigured for them to be paired. Once both channels are reconfigured, the input channel is set in the occupied status and both the input and output channels’ mapping are updated to save the new channel mapping details. Next, the output channel object (and the new partner input channel, in case the input channel is associated to the same subsystem) are stored to the InterfaceManager for the channel routing function to consider the pairing of both channels. Finally, the CD descriptor of the CD that uses the output channel is stored in the Scheduler instance for it to be included for scheduling. In case the CD that uses the input channel belongs to the same subsystem, the CD descriptor of the CD that uses the input channel will be stored in the Scheduler instance (in case the CD is included for scheduling) or in the CDOObjectsBuffer (in case the CD is not included for scheduling).

In case the current partner input channel is not associated to the same subsystem (in part (b) of Figure 6.5), the method finds the name of the subsystem the current partner input channel is associated to (‘partner subsystem’). Once it’s found, the method finds the physical address of the
partner subsystem, then it checks whether any links which can be used to communicate with the partner subsystem are present. If no such links are present, the method initiates link creation process by invoking the ExecuteLinkCreation() method of the LinkCreationSender class. If the ExecuteLinkCreation() method has successfully created a new link, the method transmits a query to the RTS of the partner subsystem to query whether the partner input channel which is to be bound with the output channel is in the occupied status (sent via the new link). If the RTS of the partner subsystem transmits back a response which indicates that the to-be-bound partner input channel is in the occupied status, the method inserts a new entry to the CDLCBuffer using the AddInvStatChanReconfig() method which can be accessed by the consumer CD to notify the consumer CD that the channel reconfiguration is not performed due to the input channel is in the occupied status. Otherwise, if the to-be-bound partner input channel is not in the occupied status, the method proceeds to reconfigure the output channel to be paired with the new partner input channel. After the output channel has been reconfigured, the output channel instance is stored to the InterfaceManager to include the new changes to the channel routing function of the InterfaceManager. Then, the method stores the new mapping of the output channel using the AddInvStatChanReconfig() method which can be accessed by the consumer CD to inform the consumer CD that the channel reconfiguration has been performed. Finally, the method stores the CD descriptor of the consumer CD to the Scheduler instance. The code snippet showing the implementation of the ReconfigInvServChannel() can be found in [177].

6.2.3 CD Termination

CD Termination process follows the sequence diagram shown in Figure 6.6. Referring to Figure 6.6, during the execution time, CD invokes the SJ.KillCD() of the Dynamic Function Call (the Dynamic Function Call is implemented as a Java class named SJ which contains the function calls described in Table 6.1). When the SJ.KillCD() method is invoked, it stores the name of the CD to be terminated (which was passed by the SJ.KillCD() method) to the CDLCBuffer using the TransferRequestKillCDToBuffer() method. Then, during the reconfiguration time, the SystemJProgram class checks the CDLCBuffer if a request to perform CD termination exists. If a request exists, the SystemJProgram class obtains the Scheduler instance stored in the class which contains all the CDs scheduled for execution (GetScheduler()) and the InterfaceManager instance (GetIM()) stored in the IMBuffer class. Next, the functionality represented as the RunTerminate() is invoked to remove signals and channels, any occurring channel rendezvous associated to the CD to be removed, and the CD descriptor of the CD. The RunTerminate() updates the Scheduler and InterfaceManager (shown as IM) instances. Then, the service description, the CD class file, and the CD configuration are removed. Next, the updated Scheduler and InterfaceManager
Results and Discussions

instances are set to be used by the RTS to run CDs in the subsystem. Finally, the InterfaceManager instance is stored in the IMBuffer, and the SystemJProgram class triggers the transmission of Advertisement by the SOA RTS.

![Diagram of CD termination sequence](image)

Figure 6.6. CD termination sequence diagram.

The RunTerminate() method depicted in Figure 6.6 complies with the process flowchart shown in Figure 6.7. Referring to Figure 6.7, the functionality begins by checking whether the CD is included for scheduling. If the CD is included for scheduling, the functionality excludes the CD from the scheduler and terminates all threads which handle the physical interface of the input signals. If the CD uses channels, the occupied status of all input channels (if they are in the occupied status) paired with output channels used by the CD is removed, any occurring channel blocking are preempted and their partner channels are unbounded from the channels. If the partner channels are associated to the same subsystem with the CD, the channel reconfiguration to unbind the partner channels are handled directly by the RTS, meanwhile if the partner channels are associated to different subsystems, the RTS transmits a command to the respective RTS to perform channel reconfiguration to unbind the channel pairing (as shown in the ‘Unbind Partner Channel,
Results and Discussions

Terminate Channel Blocking’ block in Figure 6.7). Then, the CD descriptor, the CD configuration, service description, and code of the CD are removed.

The implementation of the RunTerminate() is described in Listing 6.2. Referring to Listing 6.2, the functionality obtains all requests to perform CD termination (line 9). Then, the functionality invokes the removeCD() method to terminate the threads handling the physical interface of the input signals used by the CD, preempt any occurring channel rendezvous in the CD, and remove the CD descriptor of the CD. If the CD was in Active state, the “CD in Active state” part is executed, otherwise if the CD was in Suspended state, the “CD in Suspended state” part is executed. Next, the functionality removes the service description of the CD (line 20-29, indicated by the “Remove Service Description” part), finds the CD code file and deletes it from the memory (line 30-35, indicated by the “Delete CD File” part), and updates the data structure storing the configuration of all CDs in the subsystem (line 36-48, indicated by the “Update CD Configuration Data Structure” part). Once all requests to terminate CDs have been served, the functionality clears the buffer which stores the request to perform CD termination (line 49).

Figure 6.7. The process flowchart governing the functionality that handles CD termination.
The implementation of the removeCD() method can be found in [177].
6.2.4 CD Creation

CD Creation process follows the sequence diagram shown in Figure 6.8. Referring to Figure 6.8, during the execution time, CD invokes the SJ.CreateCD() provided by the Dynamic Function Call. When the SJ.CreateCD() is invoked, the method invokes the function that searches the XML files describing the CD configuration and service description associated to the CD in the memory and then parses them to obtain the CD configuration and service description (indicated by the ParseConfigAndServDesc() method). Then, the method passes and stores the CD name, CD configuration, and service description to the data structure of the CDLCBuffer (indicated by the TransferCDName() and TransferConfigAndServDesc()). During the reconfiguration time, the SystemJProgram class checks the CDLCBuffer if a request to perform CD creation is present. If the request is present, the SystemJProgram class obtains the Scheduler instance stored in the class which contains all the CDs scheduled for execution (indicated by the GetScheduler() method) and the InterfaceManager instance (indicated by the GetIM() method) stored in the IMBuffer class, a class with data structure which grants access to the InterfaceManager instance used by the RTS. Then, SystemJProgram class invokes the functionality provided by the Reconfigurator Function (indicated by the RunCreate()) to perform dynamic behaviours (in this case, to perform CD creation). The functionality also fetches the CD configuration and service description stored in the CDLCBuffer for the CD creation purpose. The functionality returns the Scheduler and InterfaceManager instance which have been updated to include the new CD. The Scheduler and InterfaceManager instances are then stored in the SystemJProgram class for use in the next execution time (indicated by the UpdateScheduler() and UpdateIM()), and then the InterfaceManager instance are stored in the IMBuffer (indicated by the SaveIM()). Finally, the transmission of Advertisement is triggered to update the list of service description stored in the global service registry with the service description of the CD (indicated by the TriggerRefreshAdv()).
Results and Discussions

The SJ.CreateCD() is realized by the CreateCD() method of the SJ class. The code snippet of the CreateCD() can be found in Listing 6.3. Referring to Listing 6.3, the CreateCD() invokes the ParseCDMap() and ParseServDesc() methods that use the XML parsing functionality of the CDLCMapParser class to parse CD configuration and service descriptions, respectively (line 4-5). Once the parsing function is completed, the CD name and the parsing results of the CD configuration and service descriptions are stored in the CDLCBuffer class (line 6-8).

Listing 6.3. The SJ.CreateCD() function call, ParseCDMap() and ParseServDesc() methods.

```java
1   public class SJ {
2      //...further class descriptions...//
3  public static synchronized void CreateCD(String CDName, String filenameCDMapXML, String filenameCDServDescXML) {
4            JSONObject CDMap = ParseCDMap(CDName, filenameCDMapXML);
5            JSONObject CDServDesc = ParseServDesc(filenameCDServDescXML);
6            CDLCBuffer.TransferRequestCreateCDToBuffer(CDName);
7            CDLCBuffer.AddTempSigChanMapCD(CDMap);
8            CDLCBuffer.putUpdateServiceDescription(CDServDesc);
9        }
10    }
11   public static JSONObject ParseCDMap(String CDName, String filename){
```

Figure 6.8. CD creation sequence diagram.
Results and Discussions

```java
CDLCMapParser cdpars = new CDLCMapParser();
JSONObject js = new JSONObject();
try {
    js = cdpars.parse(CDName, filename);
} catch (Exception ex) {
    ex.printStackTrace();
}
return js;
}

public static JSONObject ParseCDMap(String CDName, String filename){
    CDLCMapParser cdpars = new CDLCMapParser();
    JSONObject js = new JSONObject();
    try {
        js = cdpars.parse(CDName, filename);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return js;
}

public static JSONObject ParseServDesc(String filename){
    ServDescParser cdsdparse = new ServDescParser();
    JSONObject js = new JSONObject();
    try {
        js = cdsdparse.parse(filename);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return js;
}

//....further class descriptions...
```

After both parsing methods are invoked (line 4-5), the TransferRequestCreateCDToBuffer() method of the CDLCLbuffer class is invoked to store the CD name to be created, followed by the AddTempSigChanMapCD() method to store the CD configuration of the CD, and the putUpdateServiceDescription() method to store the service description associated to the CD. During the reconfiguration time, the SystemJProgram class invokes the functionality of the Reconfigurator Function to create the CD which complies to the process flowchart shown in Figure 6.9. Referring to Figure 6.9, the functionality begins by identifying whether a CD with the same name (as the CD name put in the CD creation request) is present. If a CD with the same name is present, the functionality executes the removeCD() method. Next, the functionality obtains the CD configuration and service description, and create the CD descriptor. Once the CD descriptor is created, the functionality executes the AddCD() method. The method starts by instantiating all environment signals and the threads handling the physical interface of the input signals. Next, it instantiates input and output channel of the CD based on the CD configuration and stores the CD.

---

16 To ‘replace’ the CD with new updated behaviour.
Results and Discussions
configuration and service description. The inclusion of the CD in the scheduler depends on whether it is a new CD, or the previous CD’s (the CD with the same name) macro-state before the CD creation is initiated. If the created CD is a new CD or if the previous CD’s macro-state is in the Active state, the CD is included to the scheduler for execution in the next tick, otherwise the CD is not included in the scheduler. Finally, the CD service description is stored in the internal service registry.

Figure 6.9. The process flowchart of the functionality denoted by the RunCreate().

The “Instantiate Each Input (or Output) Channel & Link (if needed)” blocks shown in Figure 6.9 are further detailed in the flowchart shown in Figure 6.10. For input channel, if the partner output channel which will be bound with the input channel belongs to a different subsystem, the functionality checks whether any links are available to pass data to the partner output channel. If no links are available, the functionality initiates the link creation process and stores the link configuration once the link has been created (as shown in Figure 6.11). Then, the functionality
Results and Discussion

queues a command to be sent to the RTS of the partner output channel via the created link to remove the pairing between the partner output channel and its partner, preempt any occurring channel blocking on the partner of the partner output channel, bind the partner output channel with the input channel, and stores the new mapping of the partner of the partner output channel. In case the partner output channel which will be bound to the input channel belongs to the same subsystem with the input channel, the functionality detects in which subsystem the partner of the partner output channel belongs to. If the partner of the partner output channel belongs to a different subsystem with the input channel, the functionality queues a command to be sent to the RTS of the partner of the partner output channel via link to remove the binding of the partner of the partner output channel with the partner output channel, preempts any channel blocking of the partner of the partner output channel, and stores the new mapping of the partner of the partner output channel. If the partner of the partner output channel belongs to the same subsystem, the functionality reconfigures the partner of the partner output channel to remove its binding with the partner output channel, preempts occurring channel blocking of the partner of the partner output channel, resets the occupied status, and stores the mapping of the partner of the partner output channel. Then, the functionality proceeds with preempting channel blocking of the partner output channel, binding the partner output channel to the input channel, and stores the new mapping of both the input channel and the partner output channel.

For output channel, the functionality checks whether the partner input channel which will be bound with the output channel and the output channel belongs to two different subsystems. If the partner input channel doesn’t belong to the same subsystem with the output channel, the functionality checks whether any links are available to pass data to the partner input channel. If no links are available, the functionality initiates the link creation process and stores the link configuration once the link has been created (as shown in Figure 6.11). After the link is created, the functionality queues a command to be sent to the RTS of the partner input channel via the created link to unbind the partner input channel from its partner (if any), preempt any occurring channel blocking of the partner of the partner input channel, reset the occupied status of the partner input channel, bind the partner input channel with the output channel, and stores the new mapping of the partner of the partner input channel. In case the partner input channel which will be bound to the output channel belongs to the same subsystem with the output channel, the functionality searches the subsystem which the partner of the partner input channel belongs to. If the partner of the partner input channel belongs to a different subsystem with the output channel, the functionality queues a command to be sent to the RTS of the partner of the partner input channel via link to remove the binding of the partner of the partner input channel with the partner input
Results and Discussions

channel, preempt any channel blocking of the partner of the partner input channel, and stores the new mapping of the partner of the partner input channel. If the partner of the partner input channel belongs to the same subsystem, the functionality removes the binding of the partner of the partner input channel with the partner input channel, preempts occurring channel blocking of the partner of the partner input channel, and stores the mapping of the partner of the partner input channel. Then, the functionality proceeds with preemting channel blocking of the partner input channel, binding the partner input channel to the output channel, and stores the new mapping of both the output channel and the partner input channel.

Figure 6.10. The “Instantiate Each Input (or Output) Channel & Link (if needed)” blocks.
Results and Discussions

The implementation of the functionality handling CD creation (based on the flowchart shown in Figure 6.9) is described in Listing 6.4. Referring to Listing 6.4, the functionality is described in line 2-59. The functionality starts by creating an instance of the ClockDomainLifeCycleSignalImpl class, a Java class which contains the functionality to instantiate signals and channels based on the provided CD configuration. Then, it obtains the CD configuration (line 4-5) and the name of the subsystem the CD belongs to (line 6). Next, the functionality obtains all of the CD names to be created (line 8) and their service description (line 9).

After that, the CD descriptors are created from the CD class files with the class file names specified in the CD configuration, which after that the CD descriptors are retained temporarily (line 11-19, indicated by the “Create CD Descriptor” part). Once the CD descriptor creation process is finished, the functionality begins the signal and channel instantiation. In case a CD with the same name is already present in the subsystem and was put in the Active state before the CD creation begins (line 22), the functionality invokes the removeCD() method of the ClockDomainLifeCycleSignalImpl class to remove the CD descriptor of the existing CD (line 23). Next, the functionality initialises and configures the signals and channels of the new CD using the AddCD() method of the ClockDomainLifeCycleSignalImpl class (line 26-30). This scenario is indicated by the “Existing CD in Active state” part.

![Flowchart](image_url)
Results and Discussion

Listing 6.4. The functionality represented as the RunCreate().

```java
public class ClockDomainLifeCycleManager{
    public Vector run(Scheduler sc, InterfaceManager im){
        ClockDomainLifeCycleSignalImpl cdlcmpl = new ClockDomainLifeCycleSignalImpl();
        JSONObject jCurrMap = SJSSCDSignalChannelMap.getJCurrentSignalChannelMapping();
        JSONObject jLocalCDs = jCurrMap.getJSONObject(keyCurrSS);
        String keyCurrSS = SJSSCSignalChannelMap.getInstance();
        //...further method description...
        Vector allCreateCDs = CDLCBuffer.getAllRequestCreateCD();
        JSONObject jsCDSD = CDLCBuffer.GetTempUpdateServDesc();
        HashTable CrcDs = new HashTable();
        for(int i=0;i<allCreateCDs.size();i++){
            String CDNName = allCreateCDs.get(i).toString();
            JSONObject CDDet = CDLCBuffer.GetTempSigChanMap(CDNName);
            String sclsz = jsCDS.getS(String("CDClassName"));
            ClockDomain newCD = (ClockDomain)CrrCDs.forName(sclsz).newInstance();
            CrcDs.put(CDNName, newCD);
        }
        for(int i=0;i<allCreateCDs.size();i++){
            String CDNName = allCreateCDs.get(i).toString();
            if(CrcDs.containsKey(CDNName)){
                Vector vec = cdlcmpl.removeCD(jsLocalCDs, keyCurrSS, CDNName, "Active", im, sc);
                im = (InterfaceManager)vec.get(0);
                CDLCBuffer.GetTempSigChanMap(CDNName);
                newCD.setName(CDNName);
                newCD.setState("Active");
            } else if(CDOBJECTSBuffer.containsKey(CDNName)){
                Vector vec = cdlcmpl.removeCD(jsLocalCDs, keyCurrSS, CDNName, "Sleep", im, sc);
                im = (InterfaceManager)vec.get(0);
                CDLCBuffer.GetTempSigChanMap(CDNName);
                newCD.setName(CDNName);
                newCD.setState("Sleep");
            } else {
                Vector vec = cdlcmpl.addCD(jsLocalCDs, CDDet, keyCurrSS, CDNName, newCD, CrcDs, true, im, sc);
                im = (InterfaceManager)vec.get(0);
                CDLCBuffer.GetTempSigChanMap(CDNName);
            }
        }
        return;
    }
}
```
Results and Discussions

In case a CD with the same name is already present in the subsystem and was put in the Suspended state before the CD creation begins (line 33), the functionality also invokes the removeCD() method of the ClockDomainLifeCycleSignalImpl class to remove the CD descriptor of the existing CD along with uninitialising its signals and channels (line 34). Then the functionality initialises and configures the signals and channels of the new CD using the AddCD() method of the ClockDomainLifeCycleSignalImpl class (line 37-41). This scenario is indicated by the “Existing CD in Suspended state” part. Finally, if there is no existing CD with the same name in the subsystem, the functionality proceeds with the initialization and configuration of the CD’s signals and channels using the AddCD() method of the ClockDomainLifeCycleSignalImpl class (line 45-49). This scenario is indicated by the “New CD” part.

Once the initialization of signals and channels process is finished, the buffer which temporarily stores the CD configuration of the created CDs is cleared (line 53). The service description of the created CDs is stored in the local service registry (line 55). Also the buffer which temporarily stores the service description of the created CDs is cleared (line 56), followed by the buffer which contains the name of the created CDs (line 57). The RTS uses the CDObjectsBuffer class to temporarily store CD descriptors. The implementation of the AddCD() method can be found in [177].

6.2.5 CD Suspension

CD Suspension follows the sequence diagram shown in Figure 6.12.

![Figure 6.12. CD suspension sequence diagram.](Image)
Results and Discussion

Referring to Figure 6.12 and Listing 6.5, when a CD invokes the SJ.SuspendCD() method during the execution time (line 1-3), the method passes the name of the CD to be suspended to be stored to the CDLCBuffer class using the TransferRequestSuspendCDToBuffer() method. During the reconfiguration time, the SystemJProgram checks whether a request to suspend CD exists. Once the SystemJProgram obtains all of the requests to suspend CD (if any), the SystemJProgram obtains the Scheduler and InterfaceManager instances to access all CDs included for scheduling. Then, the SystemJProgram invokes functionality of the Reconfigurator Function (implemented as the ClockDomainLifeCycleManager class in the RTS) represented as the RunSuspend(), which is shown in line 5-12. Once the RunSuspend() is executed, the scheduler is updated to remove the suspended CD from scheduling, and then the transmission of Advertisement is triggered which excludes the service(s) of the suspended CD (i.e. the service(s) are considered ‘unavailable’). The functionality described in line 5-12 of Listing 6.5 complies with the sequence diagram shown in Figure 6.13. The disableCD() method, which excludes CD from scheduling and stores data and execution state, invoked in line 9 follows the process flowchart shown in Figure 6.13.

![Diagram of disableCD() method process flowchart within the RunSuspend()](image)

Figure 6.13. The disableCD() method process flowchart within the RunSuspend()
Results and Discussions

Listing 6.5. The code snippet describing the SJ.SuspendCD() and the function represented as the RunSuspend().

```
public static synchronized void SuspendCD(String CDName){
    CDLCBuffer.TransferRequestHibernateCDToBuffer(CDName);
}

public Vector run(Scheduler sc, InterfaceManager im){
    Vector CDToSuspend = CDLCBuffer.GetAllRequestHibernateCD();
    for(int i=0;i<CDToSuspend.size();i++){
        String keyCDName = CDToSuspend.get(i).toString();
        if (sc.SchedulerHasCD(keyCDName)){
            sc = cdlcimpl.disableCD(jsLocalCDs, keyCurrSS, keyCDName, sc);
        }
    }
    CDLCBuffer.ClearRequestHibernateCD();
   //…further method description…//
    }
```

Referring to Figure 6.13, the method obtains the input signals mapping from the CD configuration. Then, the method accesses individual input signals used by the CD and terminates all running threads which handle the physical interface of the input signals. Next, the CD descriptor is removed from the Scheduler instance (to be excluded from execution in the next execution time) and stored in the CDOBJectsBuffer data structure, and finally, the new macro-state of the CD is updated. The disableCD() method iterates the same functionalities for each CD included for CD suspension. The code snippet showing the implementation of the disableCD() method can be found in [177].

6.2.6 CD Resumption

CD Resumption follows the sequence diagram shown in Figure 6.14. Referring to Figure 6.14, when a CD invokes the SJ.WakeUpCD() method during the execution time, the method passes the name of the CD to be resumed (for execution) to be stored to the CDLCBuffer class using the TransferRequestWakeUpCDToBuffer() method. During the reconfiguration time, the SystemJProgram obtains the Scheduler and InterfaceManager instances. Once all requests to perform CD resumption are obtained, the functionality in the Reconfigurator Function (ClockDomainLifeCycleManager class) represented as the RunResume() method is invoked. The RunResume() method updates the Scheduler instance with the CD to be resumed to be included for execution. Then, the SystemJProgram sets the Scheduler instance to be used by the RTS for the next execution time, and then the SystemJProgram triggers the transmission of Advertisement. The WakeUpCD() method of the SJ class is shown in line 1-3 of Listing 6.6.
Referring to Listing 6.6, once the name of the CD is passed to the RTS, the dynamic behaviour handling functionality described in line 5-13 of Listing 6.6 is invoked. The functionality described in line 5-13 of Listing 6.6 complies with the sequence diagram shown in Figure 6.14. The resumeCD() method, which includes suspended CD for scheduling, invoked in line 10, complies with the process flowchart shown in Figure 6.15.
Referring to Figure 6.15, the method obtains the CD descriptor of the CD to be resumed from the CDOObjectsBuffer. Next, the method obtains the input signals mapping from the CD configuration and accesses individual input signals used by the CD. Then, the method instantiates threads to handle the physical interface of the input signals. Next, the CD descriptor is included to the Scheduler instance (to be scheduled for execution in the next execution time), and finally, the new macro-state of the CD is updated. The resumeCD() method iterates the same functionalities for each CD included for CD resumption. The code snippet showing the implementation of the resumeCD() method can be found in [177].

### 6.2.7 CD Migration

CD Migration follows the sequence diagram shown in Figure 6.16.
Results and Discussions

During Execution Time (Origin SS)

During Reconfiguration Time (Origin SS)

Referring to Figure 6.16 and Listing 6.7, when a CD invokes the SJ.MigrateCD() method during the execution time, the SJ class invokes the parsing methods (indicated by the ParseConfigAndServDesc() in Figure 6.16, which represents the ParseCDMap() and ParseServDesc() methods described in Listing 6.3, both invoked in line 2-3 of Listing 6.7) to obtain the CD configuration and service description. Then, the parameters needed for the CD migration, which includes the CD name, CD configuration, and service description, are stored in the CDLCBuffer (indicated by the TransferParams() in Figure 6.16, which represents the AddRequestMigrate() in line 4 of Listing 6.7). During the reconfiguration time, the SystemJProgram checks for any requests to perform CD migration (IsRequestMigrateCDEmpty()). If a request to migrate CD is present, the SystemJProgram class accesses the Scheduler instance (GetScheduler()) to access the CD descriptor of the migrating CDs. Next, for each migrating CD, the SystemJProgram invokes the functionality of the ClockDomainLifeCycleManager (depicted as Reconfigurator Function) represented as the RunMigration() method. The RunMigration() method initiates the handshaking process as a preparation for the CD to migrate from the source subsystem and the destination subsystem. After
Results and Discussion

the invocation of the RunMigration(), the SystemJProgram updates the Scheduler instance (UpdateScheduler()) and finally, the transmission of Advertisement is triggered (TriggerRefreshAdv()).

Listing 6.7. The MigrateCD() of the SJ class and the function represented as the RunMigration().

```java
public static synchronized void MigrateCD(String CDName, String fileNameCDMap, String fileNameCDServDesc, String DestinationSS, String MigType){
    JSONObject CDMap = ParseCDMap(fileNameCDMap);
    JSONObject CDServDesc = ParseServDesc(fileNameCDServDesc);
    CDLCBuffer.AddRequestMigrate(CDName, DestinationSS, CDMap, CDServDesc,MigType);
}

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
//…further method description...//
```

The RunMigration() follows the process flowchart and sequence diagram shown in Figure 6.17, with code snippet shown in Listing 6.7. As shown in Figure 6.17, the RunMigration() begins by obtaining all requests to migrate CD (which are stored in the CDLCBuffer) (line 8 of Listing 6.7). Since all requests to migrate CD are collected during the execution time and they may be migrated to different destination subsystems, the functionality groups all requests to migrate CD with the same destination subsystem together, and then perform the migration handshake (represented as MigHandshake(), executed in line 19 of Listing 6.7) for each destination subsystem to prepare for the migration process. The MigHandshake() starts with the instantiation of a TCP/IP and a UDP/IP sockets to be used to receive confirmation from the destination subsystem whether the origin needs to transfer dependency files (which occurs during the data transfer process), and to receive migration response message from the destination subsystem, respectively. Once a query

---

17 For example, user-created Java classes, third party libraries which will be used by the CD to run in the destination.
to initiate migration message (to be sent to the destination subsystem) is created, the MigHandshake() method queries the destination subsystem (represented as IsQueryMigAvail() in Figure 6.17) whether it is available to receive migration data. If no response is received from the destination subsystem’s side, the MigHandshake() terminates and the request to migrate CD is retained\(^{18}\). Otherwise, if the destination subsystem sends a response to acknowledge the initiation of the migration process, the MigHandshake() obtains the CD descriptor of each migrating CD which will migrate to the destination subsystem (either from the Scheduler instance in case the CD was in the Active state, or from the CDObjectsBuffer in case the CD was in the Suspended state). Next, the MigHandshake() obtains the TCP/IP physical port number (which will be used to create a TCP/IP socket by the origin subsystem to transmit migration data to the destination subsystem) from the response message sent by the destination subsystem. Then, the MigHandshake() initiates the migration process by starting the MigTransferThread thread.

Meanwhile, on the destination subsystem’s side, the request to perform migration message sent from the origin subsystem’s side is received by the MigrationAndLinkReqThread thread. Upon receiving the message, the MigrationAndLinkReqThread obtains the UDP/IP and TCP/IP physical port numbers\(^{19}\) which are used by the origin subsystem to receive migration response message and receive confirmation (during the transfer process) from the destination subsystem’s side to send dependency files\(^{20}\), respectively. Then, the MigrationAndLinkReqThread thread creates a TCP/IP socket which will be used to receive migration data from the origin subsystem’s side. If the created response message indicates that the request to perform migration is acknowledged, the MigrationAndLinkReqThread initiates the MigrationRecTransferThread. Both the MigrationTransferThread (of the origin subsystem) and the MigrationTransferRecThread (of the destination subsystem) handle the transfer process. The interaction between the MigrationTransferThread and MigrationTransferRecThread is described in Figure 6.18.

---

18 Thus the RTS will make another attempt to initiate migration process will be performed in the next reconfiguration time.
19 Included in the request to perform migration message.
20 See footnote 17
Figure 6.17. The RunMigration() process flowchart (top) and sequence diagram detailing the Initiate Transfer Functionality block of the MigHandshake() with the MigrationAndLinkReqMsgThread of the destination subsystem (bottom).
Results and Discussion

For each dep class
If dep file not present in dest SS
For each CD
If weak migration
CreateCDDesc()
StoreCDDesc()
If strong migration
For each CD
CD Descriptor
StoreCDDesc()
CD Configuration
UpdateCDConfig()
SJSSCDSignalChannelMap(Destination SS)

Continued below...
Results and Discussion

Figure 6.18. The MigTransferThread and MigTransferRecThread interaction diagram.
Results and Discussions

Referring to Figure 6.18, the transfer process is started with the transmission of a START message from the MigTransferThread. After the START message is sent, the MigTransferThread transmits the origin subsystem’s physical address, name, and the number of migrating CDs code files which will be transferred to the destination subsystem.

In case of strong migration, the MigTransferThread transmits the number of CD descriptor which will be transferred to the destination subsystem. Next, the MigTransferThread scans the CD code files to find their dependencies (e.g. required third party libraries, application-dependent Java classes). Once all dependencies are found, the thread transfers the number of dependency files of the CD to the destination subsystem, and then the thread attempts to transfer each dependency file to the destination subsystem. The thread initially transfers the path of individual dependency file to the destination subsystem, which will be responded by the destination subsystem whether the destination subsystem requires the dependency file to be transferred to the destination subsystem. If the destination subsystem notifies the origin subsystem that it needs the dependency file, the MigTransferThread transfers the dependency file. After the dependency file is transferred, the MigTransferThread transfers the CD code file.

In case of weak migration, after the MigTransferRecThread of the destination subsystem receives individual CD code file, it creates the CD descriptor and then stores it in the CDOjectsBuffer (to be instantiated during the reconfiguration time). In case of strong migration, the CD descriptors of individual migrating CD are transferred to the destination subsystem after CD code files are transferred and stored in the CDOjectsBuffer (to be instantiated during the reconfiguration time). Following the transfer of CD code files (and CD descriptors, in case of strong migration), the MigTransferThread sends the CD configuration of the migrating CDs. Once the CD configuration is received by the MigTransferRecThread, the CD configuration is stored. Next, for each CD with channels paired with partner channels associated to other subsystems, the MigTransferThread triggers the transmission of the destination subsystem’s name to the RTS of the partner channels to notify the RTS of the partner channels that the migrating CD changes the subsystem it belongs to. Also, the MigTransferThread transmits the CD name(s) of individual partner channels, the subsystem name(s) the partner channels are associated to, and the physical address(es) of the subsystem(s) the partner channels are associated to for channel routing purpose of the destination subsystem once the migrating CDs are instantiated in the destination subsystem. After they are received, the MigTransferRecThread updates the InterfaceManager instance of the

---

21 In case of weak migration, this information is not transferred since weak migration doesn’t involve the transfer of CD descriptor.

22 The implementation of the dependency scanning is limited to only dependency which has the same parent directory with the CD code file.
Results and Discussions

destination subsystem to include the CD names using the partner channels and the subsystem name(s) the CD names are associated to for channel routing purpose and also updates the RegAllSSAddr to include the physical address(es) of the subsystem(s) the partner channels are associated to. The MigTransferThread updates the InterfaceManager instance to remove the migrating CDs channels instances stored in the InterfaceManager. Then, the MigTransferThread transfers the service description of the migrating CDs to the destination subsystem.

Once the service description is transferred, the migrating CDs service description is removed from the origin subsystem, while it is stored in the destination subsystem. Next, the MigTransferThread checks individual migrating CDs if the channels they are using are paired with partner channels associated to the original subsystem. If a migrating CD has channels paired with other channels used by CDs (partner channels) associated to the same subsystem, the MigTransferThread temporarily stores the migrating CD’s name(s) that use these channels, the migration parameters (i.e. migration type and destination subsystem name), and the configuration of the migrating CDs, then the MigTransferThread sets a flag to trigger channel reconfiguration process during the reconfiguration time and wait until the channel reconfiguration process has been performed. The channel reconfiguration reconfigures the partner channels in the subsystem due to the migration of the migrating CDs to the destination subsystem.

**During Reconfiguration Time (Origin SS, Reconfig Channel)**

![Diagram of During Reconfiguration Time (Origin SS, Reconfig Channel)]
Results and Discussions

The channel reconfiguration process\textsuperscript{24} follows the process flowchart shown in Figure 6.19. Referring to Figure 6.19, the SystemJProgram checks the flag (used to trigger the channel reconfiguration process) to detect whether it has been set by the MigTransferThread during \textit{reconfiguration time}. If the flag is set, the SystemJProgram obtains the migrating CD’s name(s)

---

\textsuperscript{23} In the context of migration, the channel reconfiguration doesn’t change channel pairing (if the channel mapping is not changed). The channel reconfiguration alters the channel configuration which determines whether the RTS exchanges the channel communication statuses and data via shared memory or link.

\textsuperscript{24} See footnote no 23
that use channels pairing with partner channels in the same subsystem, their migration parameters, their CD configuration, and the CD configuration of other CDs in the subsystem. Next, the SystemJProgram executes the function depicted as the RunReconfigLocPartnerChan() in Figure 6.19 which reconfigures channels that are paired with partner channels in the origin subsystem. The RunReconfigLocPartnerChan() updates the Scheduler and InterfaceManager instances once these channels are reconfigured. The RunReconfigLocPartnerChan() follows the flowchart also shown in Figure 6.19. The method starts by obtaining the migration parameters of the migrating CDs. The method groups migrating CDs based on individual destination subsystems, then for each migrating CDs in the group, the method checks for individual input and output channels used by the CD to find their respective partner channels. Then, the method alters the channel configuration following the migration of the migrating CDs. Once all input and output channels paired with their partner channels in the origin subsystem have been reconfigured, the method removes the stored migration parameters before completing its execution.

The MigTransferThread continues once the channel reconfiguration process has finished. It removes the CD configuration of the migrating CDs from the CD Config Repository and the CD macro-state of the migrating CDs from the CD Macro-state Repo. Finally, the MigTransferThread transmits a STOP message to the MigTransferRecThread, indicating that the transfer process is terminated. Upon receiving the STOP message, MigTransferRecThread sets a flag on the RTS of the destination subsystem to indicate that a migration process has completed and there are migrating CDs ready to be instantiated. The instantiation of migrating CDs follows the sequence diagram shown in Figure 6.20. Referring to Figure 6.20, after the flag indicating the completion of the transfer process on the destination subsystem’s side is set, it will be visible by the SystemJProgram. Once the SystemJProgram detects that the flag is set, it obtains all CD descriptors of the migrating CD and their CD configurations, and it invokes the functionality denoted as InstMigCD() to instantiate individual migrating CD in the destination subsystem. The InstMigCD() updates the Scheduler and InterfaceManager instances to include the instantiated CD. After all migrating CDs have been instantiated, the flag is reset.

25 See footnote no 23.
Results and Discussions

During Reconfiguration Time (Destination SS, after transfer is completed)

The process flowchart governing the InstMigCD() is shown in Figure 6.20. Initially, the CD descriptor, CD configuration, and service description are obtained. Then, the signals, the threads handling the physical interface of the input signals, input channels, and output channels are instantiated. In case of strong migration, the migrating CD is returned to the macro-state before the CD goes to the Migrating state, hence if the CD was in the Active state, the CD is included in
Results and Discussions

for scheduling, or else if the CD was in the Suspended state, the CD descriptor is stored in the CDOObjectsBuffer. In case of weak migration, the migrating CD is included for scheduling. The “Instantiate Each Input Channel & Link” and “Instantiate Each Output Channel & Link” blocks shown in Figure 6.20 comply with the process flowchart shown in Figure 6.21.

![Flowchart](image_url)

Figure 6.21. Migrated CD’s channels instantiation in the destination subsystem.

The process flowcharts shown in Figure 6.10 and Figure 6.21 have the same flow, except the part shown in red colour in Figure 6.21. In case of weak migration, since CD execution starts from the beginning after it has migrated, any blocking in the partner channels of the channels used by migrating CD is terminated (preempted) and the partner channel communication statuses are reset.

155
Results and Discussion

Meanwhile, since channel communication are resumed from the point before migration is initialised in strong migration, any blocking in the partner channels of these channels are not preempted.

6.2.8 Utilizing the SOSJ Dynamic Function Calls

To demonstrate how SOSJ dynamic function calls are use, Listing 6.8 shows the code snippet of CDs describing parts of the manufacturing system shown in Figure 1.1. The code snippet describes the PECD (line 24-40) which provides the photo eye sensing service (to detect bottles on conveyors), the CBCD (line 43-58) which provides the conveyor service for bottle delivery, and the DBCD (line 1-22) which invokes SOSJ dynamic function calls to trigger dynamic behaviours.

Listing 6.8. Code snippet showing how to use SOSJ dynamic function calls.

```java
1  DBCD(
2      input String signal Comm;
3            //…Further signal and channel description...//
4  )->{
5      //…further behaviour description…//
6      SJ.CreateCD("PECD","pecd1conf.xml","pecd1sd.xml");
7      SJ.CreateCD("CBCD","cbcd1conf.xml","cbcd1sd.xml");
8      pause;
9      //…further behaviour description…//
10     SJ.SuspendCD("PECD");
11     pause;
12     //…further behaviour description…//
13     SJ.WakeUpCD("PECD");
14     pause;
15     //…further behaviour description…//
16     SJ.KillCD("PECD");
17     pause;
18     //…further behaviour description…//
19     SJ.MigrateCD("CBCD", "", "cbcd1confnew.xml","cbcd1sdnew.xml","SS2","strong");
20     pause;
21     //…further behaviour description…//
22   }
23
24  PECD(
25      input String signal PESensor
26      input String channel RecInvPE1Ch;
27      output String channel RespInvPECh;
28  )->{
29     { 
30       while(true){
31         receive RecInvPE1Ch;
32         String recvMsg = (String)#RecInvPECh;
33         await (PESensor);
34         String p1val = (String)#PE1Sensor;
35            //…further behaviour description….//
36         send RespInvPE1Ch(respMsg);
37         pause;
```
As seen in Listing 6.8, the DBCD is described in line 1-22. The SJ.CreateCD() method is invoked in line 6 and 7 to trigger the creation of CDs with name PECG and CBCD, with CD configuration described in XML files named pecd1conf.xml and cbcd1conf.xml, and service description in XML files named pecd1sd.xml and cbcd1sd.xml, respectively. The SJ.SuspendCD() method is invoked in line 10 to temporarily stop the execution of PECG, while the SJ.WakeUpCD() method is invoked in line 13 to resume the execution of suspended PECG. To terminate the execution of PECG permanently, the SJ.KillCD() method is invoked in line 16. To migrate CBCD, the SJ.MigrateCD() method is invoked in line 19, with new CD configuration described in cbcd1confnew.xml and cbcd1sdnew.xml, with the destination subsystem named SS2 and of strong migration.

6.3 Concluding Remarks

This chapter has described parts of the implementation of the SOSJ framework which are responsible for handling dynamic behaviours. They extend the SOSJ RTS and include the mechanisms to perform link creation, channel reconfiguration, CD creation, CD suspension, CD resumption, CD termination and CD migration. A set of APIs have been defined that allow the triggering of functionalities that perform dynamic behaviours and an example has been presented to show how the APIs can be used. The next chapter (Chapter 7) will describe comparisons and evaluations that show the capabilities and performance of SOSJ in addressing the programming requirements of DISS.
To showcase the capabilities and performance of SOSJ in satisfying the design and programming requirements of DISS, a number of benchmarks are run [179] [178] [180] to compare SOSJ against two state of the art programming approaches, a SOA-based WS4D JMEDS (Web Services for Devices Java Multi Edition DPWS Stack) [105], which adopts the Device Profile for Web Service (DPWS) standard [181], and a MAS programming approach JADE which possess a set of similar capabilities with SOSJ in regards to dynamic behaviours handling. The WS4D JMEDS is chosen because it is one of the most widely used SOA-based approaches, and is also based on Java (like SOSJ), thus having the same advantages (computing platform-agnostic) and drawbacks (e.g. automatic garbage collection) with SOSJ due to the use of JVM. JADE is opted not only because it is based on Java, but also it is one of the most widely used MAS-based approaches and capable of handling dynamic behaviours to certain extent. This chapter starts by describing the qualitative comparison between SOSJ, JADE, and WS4D JMEDS in terms of their capability in addressing the design and programming requirements of DISS. Then, a number of benchmarks which are run to demonstrate and compare the performance of SOSJ against JADE and WS4D JMED are specified in this chapter. Some benchmarks are carried out with certain variations in the experimental setup in order to characterize the performance of individual framework features and gain information which would not be obtainable without including these variations in the experimental setup. The results gathered from the executed benchmarks are described and discussed.

This chapter is broken down into four sections. Section 7.1 compares SOSJ qualitatively against the two aforementioned state of the arts. Section 7.2 discusses the benchmark results which showcase the performance of SOSJ in handling SOA functionalities. Section 7.3 discusses the benchmark results which showcase the performance of SOSJ in handling dynamic behaviours. Finally, Section 7.4 compares the performance of SOSJ in handling SOA functionalities and dynamic behaviours against the performance of WS4D JMEDS and JADE.
7.1 Comparison in Satisfying the Design and Programming Requirements of DISS

The capability of SOSJ and WS4D JMEDS to satisfy the design and programming requirements can be summarized in Table 7.1.

Table 7.1. The capability of SOSJ, JADE, WS4D JMEDS in satisfying the design and programming requirements.

<table>
<thead>
<tr>
<th>Design and Programming Requirements</th>
<th>WS4D-JMEDS</th>
<th>JADE</th>
<th>SOSJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Correctness</td>
<td>Limited</td>
<td>Limited</td>
<td>Supported</td>
</tr>
<tr>
<td>Reactivity</td>
<td>Limited</td>
<td>Partial</td>
<td>Supported</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Asynchronous only</td>
<td>Asynchronous only</td>
<td>Synchronous &amp; Asynchronous</td>
</tr>
<tr>
<td>Composability</td>
<td>Two-level</td>
<td>Hierarchical</td>
<td>Hierarchical</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Java-enabled, additional middleware may be necessary</td>
<td>Java-enabled, additional middleware may be necessary</td>
<td>Java enabled, no additional middleware necessary</td>
</tr>
<tr>
<td>Data Driven Computation</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Dynamicity</td>
<td>Behaviour Creation</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Behaviour Suspension</td>
<td>Limited</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Behaviour Resumption</td>
<td>Limited</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Behaviour Termination</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Behaviour Migration</td>
<td>Strong</td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Weak</td>
<td>Limited</td>
</tr>
</tbody>
</table>

As a SOA approach, the WS4D JMEDS doesn’t have any underlying formal MoC and lacks any reactive constructs, hence limit the capability of WS4D JMEDS to satisfy the functional correctness and reactivity requirements. The WS4D JMEDS allows one machine platform to describe multiple DPWS devices, in which each DPWS device is capable to host multiple services.
Results and Discussion

The use of Java allows WS4D to perform data computational process and certain degree of abstraction to run on different type of Java-capable execution platforms with little to no changes necessary by programmers, however programmers are left to their own responsibility to deal with low level implementation details of physical I/O and communication interfaces specific required for their applications, thus having additional middleware layer may be preferable or even necessary. Being based on the SOA paradigm enables WS4D JMEDS to satisfy the behaviour awareness requirements, the WS4D JMEDS also support dynamic creation and termination of software services, however no built-in mechanisms to handle dynamic suspension, resumption, and migration of software services. In contrast, SOSJ support functional correctness and reactivity using SystemJ’s built in reactive constructs based on formal semantics and MoC. Compared to WS4D JMEDS which is capable of handling asynchronous concurrency only, SOSJ is capable of handling both asynchronous and synchronous concurrency. Being based on SystemJ, SOSJ also enables programmers to focus on designing their software behaviours on higher level of abstraction, with low-level implementation details handled by the RTS. The use of Java also allows for deployment on different types of Java-enabled execution platforms with little or no changes necessary and performing data-oriented computations. Also, apart from supporting behaviour awareness through the SOA paradigm-based features, SOSJ provides the features to perform behaviour creation, suspension, resumption, termination, and also migration.

As a MAS-based approach, JADE is not based on any formal semantics and MoC, hence the responsibility of ensuring correct behaviour functionality is given to the programmers (functional correctness). While JADE is designed to have its agents react to the environment upon receiving inputs from the environment, the lack in reactive constructs limits JADE’s ability in satisfying the reactivity requirement. JADE is designed to be able to run multiple concurrent agents, where all agents are considered asynchronous in nature. Also, hierarchical software behaviour composition is possible in JADE by allowing individual agents to contain multiple software behaviours called behaviours (in JADE terminology). JADE doesn’t provide any abstracted mechanisms for agents to communicate with the environment. In certain cases, an additional middleware may be necessary to relieve programmers’ burden from dealing with low-level implementation and physical communication details. Also, data-driven computation in JADE is enabled by using Java which is intrinsic in JADE. To satisfy the dynamicity requirement, JADE provides the features to handle dynamic creation, suspension, resumption, termination, and migration of agents. However in regards to migration, JADE only allows for strong migration.

In contrast, SOSJ is based on formal semantics and GALS MoC in addition to built-in reactive constructs to satisfy functional correctness and reactivity requirements, respectively. Compared to
Results and Discussions

JADE, SOSJ is able to handle both asynchronous and synchronous concurrency. Hierarchical software composition in SOSJ is also allowed through multiple concurrent CDs in a subsystem, with each CD permitted to have hierarchical synchronous reactions. SOSJ RTS handles low-level implementation and physical communication, thus allowing programmers to focus on the software behaviours at the system-level. Similar to JADE, SOSJ is also capable to perform data-driven computation using Java. In addition, SOSJ is also capable to handle dynamic creation, suspension, resumption, termination, and migration. However, unlike JADE which only allows strong migration, SOSJ allows both strong and weak migration.

7.2 Experimental Results: SOA Handling of SOSJ

A set of benchmarks are created and run to evaluate and compare the performance of the SOA functionalities of SOSJ with a SOA-based programming approach.

7.2.1 Performance in Handling Service Discovery

The following benchmarks are run to show the performance of SOSJ in handling Discovery and Discovery Reply message traffic.

7.2.1.1 Benchmarks

The benchmarks consider the deployment of 5-50 identical subsystems on distributed platforms with varying length of Advertisement expiry time from 4-50 seconds. Note that changes in expiry time affect Advertisement refresh rate, which affects the overall SOA traffic the global service registry needs to handle. Each subsystem contain 5 CDs which govern the behaviour of 3 conveyors, 1 diverter, and 1 photo eye. (e.g. CB1, CB2, CB3, diverter D1, and PE1 sensor, referring to Figure 1.1).

7.2.1.2 Experiment Setup

The experiments are performed on 1-5 Beaglebone Black (BB) embedded platforms (1 GHz single core ARM Cortex A8 processor with 512 MB RAM), with each BB running a maximum of 10 subsystems, while the global service registry application is run on a Raspberry Pi Model 2B (RPI2B) platform (900 MHz quad core ARM Cortex A9 processor with 1 GB RAM). The BB and RPI2B platforms run Linux OS. Discovery message is sent and Discovery Reply is received by a PC machine (Intel i5 quad core 1.9 GHz with 8 GB RAM) which collects the presented data using Wireshark packet sniffer application. These computing platforms are interconnected through a IPv4-based physical network. The physical layout of the experimental setup is illustrated in Figure 7.1.
Results and Discussions

Figure 7.1. Layout of the Discovery – Discovery Reply experimental setup

7.2.1.3 Results

The timestamps of when a Discovery message is sent and when the Discovery Reply message sent as a response to the Discovery message is received are logged. The Average Round Trip Time (Avg RT Time) of 100 Discovery-Discovery Reply packets are calculated. The results are presented in Figure 7.2. Each range bar shown in Figure 7.2 spans from the average + standard deviation (m+s) and average – standard deviation (m-s) value.

Figure 7.2. SOSJ Discovery – Discovery Reply Avg RT Time with different Advertisement expiry time.
Results and Discussions

From Figure 7.2, it is clear that as the number of subsystems in a SOSJ program increases, the time needed for a Discovery Reply to arrive after a Discovery is sent increases almost linearly. However, changes in expiry time appear to have little effect on the average round trip time. It is likely that changes in message traffic due to changes in expiry time have much less effect in altering the round trip time compared to the one caused by changes in the number of subsystems. Since Advertisement is sent per RTS, changes in the number of CDs in each subsystem do not affect the average round trip time.

7.2.2 Performance in Handling Request for Advertisement – Advertisement

The following benchmarks are run to show the performance of SOSJ in handling Request for Advertisement and Advertisement message traffic.

7.2.2.1 Benchmarks

The benchmarks consider the deployment of a single subsystem and the global service registry application running on two separate machines. The subsystem contains multiple identical conveyor CDs which provide conveyor services and have their service descriptions advertised. The number of CDs in the subsystem is varied from 25-100, with the advertisement expiry time in 5-50 seconds range.

7.2.2.2 Experiment Setup

The subsystem and the global service registry application are deployed on one Beaglebone Black (BB) platform running Linux and a PC (Intel i5 quad core 1.9 GHz with 8 GB RAM) running Windows, respectively. Both are interconnected through a IPv4-based network. The presented data are collected using Wireshark application. The layout of the experiment setup is illustrated in Figure 7.3.

Figure 7.3. Layout of the Request for Advertisement – Advertisement experimental setup.
Results and Discussions

7.2.2.3 Results

The timestamps of when a Request for Advertisement message is sent and when the responding Advertisement message is received are logged. The Average Round Trip Time (Avg RT Time) of 100 Request for Advertisement - Advertisement is calculated. The variations in the number of CDs and Advertisement expiry time are considered to observe whether the difference in the number of services and Advertisement transmission period affects the Avg RT Time. The result of the benchmark is shown in Figure 7.4. Each range bar shown in Figure 7.4 spans from the average + standard deviation (m+s) and average – standard deviation (m-s).

![Request for Advertisement - Advertisement Average Round Trip Time](image)

Figure 7.4. SOSJ Request for Advertisement - Advertisement average round trip time with different number of CDs and Advertisement expiry time.

As shown in Figure 7.4, it is clear that both the number of CDs (which have their services included for advertisement) and advertisement expiry time affect the Avg RT Time. Higher number of conveyor CDs corresponds to bigger data structure to contain the service description of all conveyor CDs, which contributes to the time for the SOSJ RTS to construct Advertisement message which is sent as a reply to Request for Advertisement message. Shorter Advertisement
Results and Discussions

expiry time (i.e. more frequent Advertisement) means more Advertisement messages that need to be handled by the SOSJ RTS, leading to a delayed response (i.e. reflected as the increase in Avg RT Time) of Advertisement message transmission which is sent as a reply to Request for Advertisement.

7.3 Experimental Results: Dynamic Behaviour Handling of SOSJ

A set of benchmarks are created and run to evaluate the performance of SOSJ in handling dynamic behaviours and reconfigurations. The experiments consider several execution platforms to observe the effect of computational capabilities (e.g. processor clock rate, RAM capacity) on the performance of individual SOSJ framework features in handling dynamic behaviours.

7.3.1 Performing Link Creation

The following benchmarks are created and run to observe the performance of SOSJ in handling link creation process.

7.3.1.1 Benchmarks

The benchmark considers the creation of 10, 25, and 50 links which will be used to physically exchange channel data to CDs contained in 10, 25, and 50 subsystems deployed across multiple execution platforms.

7.3.1.2 Experiment Setup

The benchmark considers three different execution platforms, Beaglebone Black (BB), Raspberry Pi 2 Model B (RPi2B), and the Hard Processor System (HPS) of the Altera DE1 SoC (Altera DE1) which has a 800 MHz dual core ARM Cortex A7 processor and 1 GB RAM. Each of these execution platforms are used to trigger the link creation process. The subsystems are deployed across 5 BB platforms evenly, hence the deployment of 10, 25, and 50 subsystems will have 2, 5, and 10 subsystems per BB, respectively. The layout of the experiment setup is shown in Figure 7.5.
Results and Discussion

Figure 7.5. Layout of the dynamic link creation experiment setup.

7.3.1.3 Results

The total time for link creation process is recorded, and then the average total time over 10 link creation processes for 10, 25, and 50 links are calculated. The results are presented in Figure 7.6. Each range bar shown in Figure 7.6 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

![Figure 7.6. Average total time to perform link creation process in different platforms.](image)

From the results, it appears that difference in computational capabilities of the execution platforms leads to little reduction in average time to perform link creation. The chart shown in Figure 7.6 presents the sum of the average link creation request time and average link creation...
Results and Discussions

handshake time, in which the link creation request time is measured from the point when a request to perform link creation process is sent until a response (acknowledgement to proceed) from the responding party has been received while the link negotiation & instantiation time represents the time needed for the ExecuteLinkCreation() to complete execution. When there are higher number of links (to interact with more subsystems) that needs to be created, the total average link creation request time increases since the RTS needs to contact the RTS of each subsystems.

7.3.2 Channel Reconfiguration for Service Invocation

The following benchmarks are run to evaluate the performance of channel reconfiguration in various platforms with different computational capabilities, as well as to observe whether and how much the performance of channel reconfiguration is affected by the difference in platform’s processor clock rate and RAM capacity.

7.3.2.1 Benchmarks

Two scenarios are considered for the benchmarks. Both involve one CD with one reaction (with the consumer role) attempting to invoke the service offered by a reaction (running the provider role) in another CD which provides conveyor service. The first scenario considers both CDs belonging to the same subsystem, while the second scenario considers each CD belonging to two different subsystems. In the first scenario, both CDs are run on the same execution platform, while the second runs each CD on two separate execution platforms.

7.3.2.2 Experiment Setup

The BB, Raspberry Pi 2B (RPI2B), and high performance system with ARM processor in Altera Cyclone V device on Altera DE1 board are chosen as the execution platforms. The first scenario has both CDs running on either BB, RPI2B, or Altera DE1. The second scenario has the consumer CD running on either BB, Altera DE1, or RPI2B, while the provider CD running on BB platform.

7.3.2.3 Results

500 and 1000 consecutive channel reconfigurations are run for 10 times in both scenarios with the client CD running on BB, Altera DE1, and RPI2B platforms and the provider CD running on another BB platform. The benchmarks observe the total time required by the client CD side to achieve these channel reconfigurations, and then the average total time over 10 times of 500 and 1000 consecutive channel reconfigurations is calculated. The results of the first and second benchmarks are presented in Figure 7.7 and Figure 7.8, respectively. Each range bar shown in
Results and Discussion

Figure 7.7 and Figure 7.8 and spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

![Average Total Time To Perform Channel Reconfiguration (Single Subsystem)](image1)

Figure 7.7. Average total time required to perform channel reconfigurations in case of single subsystem.

![Average Total Time To Perform Channel Reconfigurations (Distributed Subsystems)](image2)

Figure 7.8. Average total time required to perform channel reconfigurations in case of distributed subsystems.
Results and Discussions

From the results presented in Figure 7.7 and Figure 7.8, it can be noticed that the overall average time required to perform channel reconfiguration in the single subsystem case is significantly lower than in subsystems distributed in the network due to additional network communication overhead. In both cases, the overall time for channel reconfiguration is spent on three different operations. The first operation is to send a request to perform channel reconfiguration to the RTS (for single subsystem) or transmission of request to perform channel reconfiguration to the RTS of the opposite subsystem (for distributed subsystems) to query whether the partner input channel (which will receive the service invocation request message) is available to be bound to the corresponding output channel used to send the service invocation request message (indicated as “Request for Channel Reconfiguration” in Figure 7.7 and Figure 7.8). The second operation is to obtain the acknowledgment to proceed with the channel reconfiguration process (for single subsystem, indicated as “Obtain Proceed Acknowledge” in Figure 7.7) or for the RTS of the corresponding subsystem to wait until a response message from the RTS of the opposite subsystem is received (for distributed subsystems, indicated as “Wait Until Response Received” in Figure 7.8). The last operation is for the RTS of the corresponding subsystem to modify the corresponding channel configuration and the channel mapping accordingly (indicated as “Modify Channel Configuration” in Figure 7.7 and Figure 7.8).

The time required for channel reconfiguration in case of distributed subsystems increases linearly following the number of channel reconfigurations. For single subsystem, most of the time to perform channel reconfiguration is spent on the ‘Modify Channel Configuration’, which occurs during the time when the execution of the reactive interface handled by the RTS or house-keeping time. For distributed subsystems, higher time in “Modify Channel Configuration” is because the RTS of the corresponding subsystem needs to exchange data on parameters needed for channel reconfiguration with the RTS of the partner subsystem (over the network) and also wait until the house-keeping time of the partner subsystem is reached for both RTS to proceed on modifying the configuration of both input and output channels. The results show that the difference in processor clock rate of DE1 and RPI2B appears to have little effect in the average total time to perform channel reconfiguration. With approximately 11% difference in processor clock rate between DE1 and RPI2B, little difference in average channel reconfiguration time is observed. However, the fact that both vary in the type of processor core (Cortex A7 on DE1 and Cortex A9 on RPI2B) may also be the cause. On the other hand, the difference in RAM capacity (of BB and DE1 or RPI2B) appears to affect the average channel reconfiguration time more than processor clock rate. Double RAM capacity in DE1 and RPI2B over BB reduces approximately to half the average channel reconfiguration time.
Results and Discussions

7.3.3 **Handling Dynamic Behaviours in SOSJ**

To assess the performance of SOSJ in handling dynamic behaviours, a number of scenarios taken from the motivating example explained in Section 1.2 are considered and used as case studies.

7.3.3.1 **Benchmarks**

The scenarios are based on the industrial manufacturing system which is part of the motivating example shown in Figure 1.1. The manufacturing system performs automatic bottle capping and storage with physical layout (which is extended from the case study in [182]) shown in Figure 7.9. A bottle is loaded by the Bottle Loader station at point B and then transported towards bottle capping and storage stations by the conveyors. During runtime, the manufacturing system may switch from Configuration 1 (where bottle is transported through point B, C, D, E, and F by CB1, CB2, and CB3) to Configuration 2 (where bottle is transported through point B and F by CB1, CB4, and CB3) and vice-versa with the addition or removal of D1 and CB4. The possible logical association of software services provided by individual stations with embedded controllers of two system configurations of Configuration 1 (blocks in black colour) and Configuration 2 (blocks in black and red colour) is also shown in Figure 7.9.

![Figure 7.9. The physical layout of the manufacturing example used for benchmarks.](image)

Based on the manufacturing example in Figure 7.9, the following scenarios are considered which cover dynamic creation, suspension, resumption, termination, and migration of CDs/agents.
Results and Discussion

to demonstrate the performance of SOSJ. These scenarios will also be used for benchmarking and comparison purposes in Section 7.4.4.

**Scenario 1:** The manufacturing system starts with Configuration 1, however during runtime, a new conveyor and diverter are introduced into the manufacturing system (CB4 and D1) and their corresponding services run on Controller 4, hence creating a new system depicted as Configuration 2.

**Scenario 2:** In either Configuration 1 or Configuration 2, the existing software behaviour in one of the stations, e.g. the Bottle Loading station, needs to be updated with new version.

**Scenario 3:** The conveyors CB1 and CB4 and the diverter D1 of Configuration 2 are controlled by separate concurrent software behaviours running on the same Controller 4. During runtime, the conveyor CB4 and diverter D1 needs to be stopped and removed temporarily for maintenance purpose (the software services of CB4 and D1 are also temporarily unavailable) without stopping CB1 software behaviour running on Controller 4. The conventional approach of switching off the physical controller in this case is not viable as it will stop the behaviour controlling the conveyor CB1. Once the maintenance operation is finished, the software controllers of CB4 and D1 will need to be restarted.

**Scenario 4:** The software behaviours governing the conveyor CB4 and the diverter D1 of Configuration 2 have been created and are running on the same Controller 4. However, during runtime it is decided to permanently stop the CB4 and D1 behaviours and remove CB4 and D1 from the manufacturing system without stopping the other software behaviours on Controller 4, hence reverting the manufacturing system from Configuration 2 back to Configuration 1.

**Scenario 5:** The software behaviours governing the conveyor CB4 and the diverter D1 of Configuration 2 have been created and are running on the same Controller 4. However, during runtime it is decided that the software behaviours of CB4 and D1 need to run on a separate hardware controller to distribute computational burden to another hardware controller. Thus, a new hardware controller named Controller 5 is introduced into the manufacturing system, and then both software behaviours migrate from Controller 4 to Controller 5 without stopping the other software behaviours on Controller 4.

**7.3.3.2 Experiment Setup**

The BB and the HPS of Altera DE1 are chosen as the execution platforms. The benchmarks measure the time in milliseconds to perform dynamic behaviours on 50 and 100 identical CDs,
Results and Discussions

each implements individual manufacturing function of Conveyor, Diverter, or Bottle Loader station.

7.3.3.3 Results

The benchmarks measure average time required for SOSJ to handle dynamic behaviours during the house-keeping time (with a minor exception to CD migration, where the migration data transfer time doesn’t exclusively occur during the house-keeping time). The average time is calculated over 10 executions of dynamic behaviours which achieve the considered scenarios. To assist in understanding the benchmark results, some information regarding the conveyor, diverter, and loader CD is presented in Table 7.2. The benchmarks consider these CDs to be independent to each other\textsuperscript{26}.

Table 7.2. Details on the Conveyor, Diverter, and Loader CD.

<table>
<thead>
<tr>
<th>CD Name</th>
<th>Number of Input Signal &amp; Channel</th>
<th>Number of Output Signal &amp; Channel</th>
<th>Lines of Code</th>
<th>Code File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conveyor</td>
<td>1 input channel</td>
<td>1 output channel, 3 output signals</td>
<td>25</td>
<td>11 kB</td>
</tr>
<tr>
<td>Diverter</td>
<td>1 input channel</td>
<td>1 output channel, 1 output signal</td>
<td>16</td>
<td>7 kB</td>
</tr>
<tr>
<td>Bottle Loader</td>
<td>1 input channel, 6 input signals</td>
<td>1 output channel, 5 output signal</td>
<td>141</td>
<td>15 kB</td>
</tr>
</tbody>
</table>

7.3.3.3.1 CD Creation & Termination

CD creation & termination are performed to achieve Scenario 1, 2, and 4. The benchmark results of the dynamic behaviours in achieving Scenario 1 and 4 are shown in Figure 7.10. Note that removeCD() denotes the method described in Section 6.2.3, while AddCD() represents the method described in Section 6.2.4, meaning that the result indicated by each method represents the average time required to execute the method. Each range bar shown in Figure 7.10 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

Based on the results, the time needed for CD creation is higher than the time needed for CD termination. The time needed for CD creation is dominated by the time needed to initialise CD configuration & service description and create the CD descriptor, with the time to create CD descriptor higher than the time needed to remove the CD code (i.e., access the memory and delete

\textsuperscript{26} Ideally, having CDs to be independent with each other grants flexibility to the programmers, allowing them to choose which CD to interact with one another, depending on their application requirements.
the CD code file), descriptor, CD configuration, and service description. Meanwhile, the time needed for CD termination is dominated by the time needed to remove the CD code, descriptor, CD configuration, and service description. Note that depending on whether the created CD needs to interact with other CDs in another subsystem and the availability of links to communicate with the partner subsystem, the overall time needed for CD creation may be longer due to link creation process. Based on the information presented in Table 7.2 and the results in Figure 7.10, it is clear that the CD code size affects the overall average time needed for CD creation. By comparing the average time needed for CD Descriptor Creation Time for the Conveyor CD and Diverter CD, the CD Descriptor Creation Time for the Conveyor CD is longer than the one for the Diverter CD, with the code file size of the Conveyor CD larger than the Diverter CD.

Figure 7.10. Average time to perform CD creation and termination to achieve Scenario 1 and 4.
Results and Discussions

Meanwhile, the benchmark results of the dynamic behaviours to achieve Scenario 2 is presented in Figure 7.11. Each range bar shown in Figure 7.11 spans from the average ± standard deviation (m+s) and average – standard deviation (m-s). “Updating” existing CD in SOSJ is possible by performing a CD creation on the existing CD, thus “overwriting” the existing CD with another one. When a CD “update” is performed, the removeCD() is executed, followed by the CD creation to re-instantiate the CD. As shown in Figure 7.11, the time required to perform CD “update” is dominated by the CD creation process, which mostly dominated by the time needed to create CD descriptor. Overall, as shown in Figure 7.10 and Figure 7.11, the use of more computationally capable platform leads to lower average time for CD creation and termination.

Figure 7.11. Average time for CD “update” to achieve Scenario 2.
Results and Discussions

7.3.3.3.2 CD Suspension & Resumption

CD suspension and resumption are performed to achieve Scenario 3. The benchmark results of the dynamic behaviours in achieving Scenario 3 is presented in Figure 7.12. Each range bar shown in Figure 7.12 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

![Average Time to Achieve CD Suspension & Resumption (Conveyor & Diverter CD) - BB and DE1](image)

Figure 7.12. Average time for CD suspension and resumption to achieve Scenario 3.

Based on the results in Figure 7.12, the time needed for CD suspension is longer than the time needed for CD resumption. The function executed during CD suspension includes the functionality that checks for and terminates any physical input signal threads used by input signals (if any) and accesses & excludes the CD from the scheduler is likely to take longer time to execute than the function executed during CD resumption. In addition, the results show that the CD code size affects the time required for CD suspension and resumption, with the average time for CD suspension and resumption of the conveyor CD longer than the average time of the Diverter CD. Similar with CD creation and termination, the use of more computationally capable platform leads to lower average time for CD suspension and resumption.
Results and Discussions

7.3.3.3.3 CD Migration

CD migration is performed to achieve Scenario 5. With SOSJ, Scenario 5 can be achieved with either strong or weak migration, thus both strong and weak migration are considered in the benchmarks. The benchmarks considers both strong and weak CD migration from one BB to another BB (BB-BB) and from a DE1 to BB (DE1-BB). The results of the benchmarks is shown in Figure 7.13. Each range bar shown in Figure 7.13 spans from the average + standard deviation \((m+s)\) and average – standard deviation \((m-s)\).

![Average Time to Achieve CD Migration (Conveyor & Diverter CD) - BB & DE1](image)

Figure 7.13. Average time for strong and weak CD migration to achieve Scenario 5.

Figure 7.13 shows that the time required for migration consists of three elements, Migration Request Time, Data Transfer Time, and Scan CD code for Dependency Time. The Migration
Results and Discussions

Request Time represents the time needed for the functionality in the RTS of the origin subsystem to transmit a request to perform migration to RTS of the destination subsystem until it receives a response from the latter, the Data Transfer Time represents the time needed for the functionality in the RTS to transfer CD code and its dependency (if any), CD descriptor (in strong migration), CD configuration, and service description, and the Scan CD code for Dependency Time represents the time required for the RTS to scan the CD code for dependency. If the RTS finds any dependency, it includes the file to be transferred to the destination if the destination requires it. It is worth noting that different from other dynamic behaviours handling processes in SOSJ, the Data Transfer Time is not exclusively occurring in the house-keeping time (i.e. the execution of other CDs in the subsystem advances as it is while the migrating CD is being transferred), since SOSJ RTS handles the data transfer concurrently to allow the execution of other CDs to advance as it is instead of having to wait until the entire CD migration process elapses. The benchmark results show the difference in Data Transfer Time between strong and weak migration, with strong migration being higher. This is due to the transfer of CD descriptor during strong migration, which doesn’t occur during weak migration. Also, the difference in CD code size clearly affect the average Data Transfer Time, with larger CD code size leading to higher average Data Transfer Time. Similarly, the Scan CD code for Dependency Time also increases when the CD code size is larger. Based on the results, the use of more computationally capable execution platform in the origin subsystem side potentially lead to lower Scan CD code for Dependency Time and the overall CD migration time, however it may increase the Data Transfer Time. This is likely because the transfer process in SOSJ is designed as synchronized process between the origin and destination, even if the origin side is naturally faster (due to the use of more computationally capable platform) than the destination, the origin side has to synchronize to wait for the destination side to be ready to advance. Hence, it is likely that the increase in Data Transfer Time in case of DE1-BB migration is caused by the migration functionality of the origin when waiting for the destination to be ready to advance.

7.4 Quantitative Comparisons

A set of benchmarks are created and run to compare the performance of SOSJ in handling dynamic behaviours and reconfigurations against JADE which is also capable to handle dynamic behaviours to certain extent.

7.4.1 Programming Framework Size

In regards to the size of the framework, even with the required third party libraries, SOSJ framework is smaller compared to JADE, as presented in Table 7.3. While JADE provides a rather
Results and Discussions

comprehensive support for handling dynamic behaviours (with exception of weak migration), the overall JADE framework size is much larger than SOSJ, which is likely due to having additional components in order to comply with the FIPA standard. In contrast, albeit having smaller size than JADE, SOSJ is capable of handling dynamic behaviours that JADE supports, and even more than JADE due to the capability of SOSJ to handle weak migration.

The original implementation of JADE framework lacks an important feature to enable agents to migrate between execution platforms/computing machines. Thus, in a scenario when agent migration between execution platforms is needed, programmers are responsible to provide the mechanisms themselves, or alternatively, use an extra third party library, e.g. the IPMS (Inter Platform Mobility Service) [186]. The library introduces mechanisms that enable JADE agents to migrate between execution platforms, however the inclusion of the library will incur an additional memory footprint of 237 kB. In contrast, SOSJ framework includes built-in mechanisms which allow CD to migrate between subsystems on the same or different execution platforms.

Table 7.3. Comparison between SOSJ and JADE in framework size.

<table>
<thead>
<tr>
<th></th>
<th>SOSJ</th>
<th>SOSJ + third-party libraries</th>
<th>JADE</th>
<th>WS4D JMEDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size kB</td>
<td>336</td>
<td>787</td>
<td>2712</td>
<td>1014</td>
</tr>
</tbody>
</table>

Also, SOSJ is smaller (even with the required third party libraries) compared to WS4D JMEDS. The SOSJ framework by itself is 336 kB, however just like SystemJ RTS, SOSJ requires the JDOM XML library [183] (150 kB in size) to extract the CD configuration (including its signal and channel mapping) which is written in XML format. In addition, since the CD configuration and service description is stored as a data structure formatted in JSON format, SOSJ requires the JSON for J2ME library (24 kB in size) and also the ASM Java byte code library [184] needed by the CD migration functionality to scan CD code to determine if the CD uses any dependencies (237 kB in size). Last but not least, the SOSJ remote reconfiguration GUI application (which allows human users to trigger dynamic behaviours through a remote machine) is 43 kB in size. Although SOSJ (along with the required third-party libraries) are smaller compared to WS4D JMEDS which is 1014 kB, SOSJ framework has the advantage in and offers wider support for handling dynamic behaviours over WS4D JMEDS.

7.4.2 Performance in Handling Service Discovery

The following benchmarks are run to compare the performance of SOSJ and WS4D JEMDS in handling service discovery message traffic.
Results and Discussion

7.4.2.1.1 Benchmarks

In terms of service discovery, the DPWS standard has the equivalent of SOSJ’s Discovery and Discovery Reply, namely Probe and Probe Match, respectively. In DPWS, Probe message is transmitted (multicast) by a client to find available DPWS devices (entities that provide services in DPWS terminology), and upon receiving the Probe message, DPWS devices respond with Probe Match message (unicast) back to the client. The benchmarks consider a scenario of 5-50 SOSJ subsystems (with 3 conveyor, 1 diverter, and 1 photo eye CDs, in total 5 CDs in each subsystem) with 4 seconds Advertisement expiry time in SOSJ, and 5-50 DPWS devices, in which each set of 5 DPWS devices represents 3 conveyors, 1 diverter, and 1 photo eye sensor.

7.4.2.1.2 Experiment Setup

The 5-50 subsystems and DPWS devices are deployed on 1-5 BB platforms, with each BB running a maximum of 10 subsystems (in SOSJ) or 10 DPWS devices (in DPWS), while the SOSJ global service registry application runs on another BB. Discovery/Probe message is sent from and Discovery Reply/Probe Match is received by a PC (Intel i5 quad core 1.9 GHz with 8 GB RAM). The PC logs the timestamps of when the Discovery message is sent and the Discovery Reply message is received (in SOSJ) and when the Probe message is sent and Probe Matches sent from all DPWS devices are received (in WS4D JMEDS) using Wireshark packet sniffer application. Based on the recorded timestamps, the average round trip time (Avg RT Time) of 100 Discovery-Discovery Reply (in SOSJ) and Probe-Probe Match (in WS4D JMEDS) is calculated. Note that in contrast to SOSJ which has a separate service registry application that receives Discovery from clients and transmits Discovery Reply to clients, in WS4D JMEDS Probe message is transmitted via multicast (one to many), with potentially multiple Probe Match replies coming from multiple devices. The layout of the experiment setup is shown in Figure 7.14.

Figure 7.14. The layout of the Discovery/Probe – Discovery Reply/Probe Match experiment setup
Results and Discussions

7.4.2.1.3 Results

The results of the benchmarks are shown in Figure 7.15. Each range bar shown in Figure 7.15 spans from the average + standard deviation (m+s) and average – standard deviation (m-s). As shown in Figure 7.15, SOSJ has lower Avg RT Time compared to WS4D JMEDS. This is due to the WS4D JMEDS runtime that waits for all DPWS devices to send Probe Match, while in SOSJ, the service consumer waits for the global service registry only to send Discovery Reply. Also, the increasing number of Probe Match messages (when more DPWS devices are involved) may prolong the round trip time.

It should be noted that SOSJ subsystems typically contain multiple CDs, where a single CD represents a single manufacturing device (e.g. conveyor device, photo eye device, etc.), while in DPWS, a DPWS device represents a single manufacturing device. Thus in the scenario, there are 25-250 CDs since each subsystems contains 5 CDs (i.e. 25-250 manufacturing devices), in comparison to only 5-50 DPWS devices (i.e. 5-50 manufacturing devices) in WS4D JMEDS. However, although WS4D JMEDS does not use centralized service registry (which removes the likelihood of single point failure) in contrast to SOSJ, Probe and Probe Matches message traffic in WS4D JMEDS may increase significantly as the number of DPWS devices increases. Further increase in Probe and Probe Match traffic can be expected when more DPWS clients perform service discovery.

Figure 7.15. Comparison of Avg RT Time of WS4D Probe – Probe Match and SOSJ Discovery – Discovery Reply.
Results and Discussion

7.4.3 Performance in Service Invocation

The following benchmarks are run to compare the performance of SOSJ and WS4D JMEDS in performing service invocation.

7.4.3.1.1 Benchmarks

Two different benchmarks are created and run: (1) both client and provider reside on a single BB and (2) the client and provider reside on two different BB execution platforms. Both benchmarks use SystemJ, SOSJ, and WS4D JMEDS frameworks, with the provider providing the photo eye sensor service and the client invoking the photo eye sensor service. SystemJ is included in the benchmark to observe the additional overhead introduced by the SOSJ RTS due to communication between client and provider (as SOSJ is developed by extending the SystemJ RTS, the overall performance overhead also includes the original SystemJ RTS’). Note that with regard to SystemJ and SOSJ, the first benchmark considers two scenarios on how the client and provider are deployed. In case of the first scenario the client and provider reside on two separate subsystems on the same machine with links connecting between the two implemented as TCP/IP sockets. The benchmark considers 500 and 1000 consecutive service invocations using SystemJ (via channel), SOSJ (via channel, and via signal for invocation within one CD in benchmark (1), or involving CDs in different SOSJ programs in benchmark (2)) and WS4D JMEDS. The total round trip time to perform each run of 500 and 1000 service invocations is measured and the average is calculated over 10 runs.

7.4.3.1.2 Experiment Setup

The benchmarks use the BB as an execution platform, with the first benchmark and the first scenario of the second benchmark using one BB and the second benchmark using two BBs.

7.4.3.1.3 Results

The results gathered from the first benchmark is presented in Figure 7.16, while the results collected from the second benchmark is shown in Figure 7.17. Each range bar shown in Figure 7.16 and Figure 7.17 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

In the first benchmark, SOSJ and SystemJ have lower average time compared to WS4D JMEDS in case of single subsystem scenario. Additional performance overhead is present in SOSJ invocation via channel due to channel reconfiguration implementation, which needs to be performed in the consumer and provider for service invocation. Meanwhile, invocation via signal
Results and Discussions

in SOSJ has even lower overhead as no rendezvous mechanism is used, albeit with no guarantee in data delivery. In SystemJ and SOSJ, when both the consumer and provider CDs belong to the same subsystem, signal and channel communication are achieved through shared memory, thus no performance overhead comes from network communication. Meanwhile in WS4D JMEDS, due to the use of DPWS communication stack that uses IP-based network, service invocation always goes through IP network, thus incurring additional overhead regardless of the execution location of the specific devices.

In case of the second scenario of the first benchmark, SystemJ and SOSJ have larger average round trip time compared to WS4D JMEDS in achieving service invocation. Higher average round trip time in SystemJ and SOSJ is caused by the fact that, apart from the transmission of service invocation request and response messages, channel communication in SystemJ and SOSJ requires rendezvous between the communicating parties, unlike WS4D JMEDS which involves only the transmission of service invocation request and response messages. On the other hand, in SOSJ, channel reconfiguration process introduces performance overhead leading to higher Avg RT Time compared to SystemJ.

![Average Total Round Trip Invocation Time](image)

**Figure 7.16** The average total round trip time of service invocations/two-way communications (single machine setting) in SystemJ, SOSJ, and WS4D JMEDS.

Similar to the second scenario of the first benchmark, from the results obtained from the second benchmark shown in Figure 7.17, both SOSJ and SystemJ channel communication have higher Avg RT Time compared to the WS4D JMEDS. This is because WS4D JMEDS transmits only service invocation request and response messages, compared to the channel communications
Results and Discussions

in SystemJ and SOSJ RTS which involve not only the service invocation request and response messages, but also the exchange of channel communication statuses (which is necessary for the rendezvous as part of the channel communication semantics). On the other hand, compared to SystemJ, additional performance overhead is incurred in SOSJ due to channel reconfiguration.

Based on the results in Figure 7.17, it is clear that significant portion of the overall invocation via channel overhead in SOSJ is due to SystemJ channel communication. Such performance reduction is a trade-off for having underlying formal semantics in the channel communication and guaranteed data delivery, which contributes to the reliability of service invocation in SOSJ. In contrast, such features are not present in WS4D JMEDS. Since SOSJ service invocation in two subsystems on one or multiple machines performed via channel involves link, the performance overhead is also affected by how links are implemented, and the overall performance overhead can be significantly reduced with different implementation of links (e.g. links implemented as UDP/IP since it does not involve the exchange of acknowledgment packets like TCP/IP). On the other hand, in this benchmark, invocation via signal in distributed machine setting goes through UDP/IP. Due to no rendezvous/handshake mechanism in signal and no need for reconfiguration process like in channel, invocation via signal offers significantly better performance compared to invocation via channel, however with less reliable service invocation due to no guarantee in data delivery. In distributed setting, SOSJ invocation via signal is shown to have comparable performance with WS4D JMEDS.
Results and Discussions

7.4.4 Performance in Handling Dynamic Behaviours

This section describes comparisons and experimental results that show the performance of SOSJ compared to JADE, a rather popular and prominently used MAS-based programming framework which possess features to handle dynamic behaviours, in handling dynamic behaviours. A number of benchmarks are created and run to compare the performance of SOSJ and JADE in handling dynamic behaviours. The same benchmarks (according to the respective dynamic behaviours to achieve the same individual manufacturing scenarios) described in Section 7.3.3.1 and 7.3.3.2 are used, except in this case, the RPI2B is used as the execution platform. Similar to the experiments in Section 7.3.3.3, the benchmarks also measure the average time required for both SOSJ and JADE to handle dynamic behaviours. The average time is calculated over 10 executions of dynamic behaviours which achieve the considered scenarios. The same CD implementation in Section 7.3.3 is used for the benchmarks, as detailed in Table 7.2.

7.4.4.1 Behaviour Creation & Termination

Behaviour creation & termination are performed in both SOSJ (CD creation & termination) and JADE (agent creation & termination) to achieve Scenario 1, 2, and 4. The benchmark results of the dynamic behaviours used to achieve Scenario 1 and 4 are shown in Figure 7.18. Each range bar shown in Figure 7.18 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$).

The overall time to perform agent creation in JADE can be split into two elements, ‘Creation of Agent Objects’ which is the time required by the JADE runtime environment to create agent descriptor to instantiate the agent, and ‘Schedule Agents for Execution’, which is the time required by the JADE runtime environment to include the instantiated agent to JADE scheduler for execution. Similar to SOSJ in terms of CD creation, the overall time to create agent descriptor dominates the overall time for agent creation in JADE. Based on the results, in regards to dynamic creation, SOSJ requires less overall average time compared to JADE due to the difference in the average time for SOSJ to create CD descriptor and execute AddCD() and for JADE to create agent objects and schedule agents for execution. Most of the difference comes from the longer time needed for JADE to instantiate agent objects. On the other hand, SOSJ performs dynamic termination slightly worse than JADE, but it can be attributed to the fact that SOSJ removes the CD code, while JADE does not involve the removal of agent code (code file) from the memory.
Results and Discussion

Figure 7.18. The average time comparison in performing behaviour creation & termination to achieve Scenario 1 & 4 – SOSJ & JADE

To achieve Scenario 2, CD creation on Loader CD is performed in SOSJ, while both agent termination and agent creation on Loader agent have to be invoked in JADE. The benchmark results of the dynamic behaviours that achieve Scenario 2 are presented in Figure 7.19. Each range bar shown in Figure 7.19 spans from the average + standard deviation ($m+s$) and average –
Results and Discussions

standard deviation ($m$-$s$). The results in Figure 7.19 also show that SOSJ’s performance surpasses JADE’s in overall average time. In addition, in SOSJ Scenario 2 can be achieved by programmers through invoking one dynamic behaviour (CD creation), while in JADE, programmers need to invoke two dynamic behaviours (i.e. agent creation and agent termination).

![Average Time to Achieve Behavior Update (Loader) - SOSJ & JADE](image)

Figure 7.19. The average time comparison in performing behaviour creation & termination to achieve Scenario 2 – SOSJ & JADE

### 7.4.4.2 Behaviour Suspension & Resumption

To achieve Scenario 3, behaviour suspension & resumption are performed in both SOSJ and JADE. The benchmark results of both behaviour suspension & resumption that achieve Scenario
Results and Discussion

3 in SOSJ and JADE are shown in Figure 7.20. Each range bar shown in Figure 7.20 spans from the average + standard deviation (m+s) and average – standard deviation (m-s).

![Figure 7.20](image_url)

Figure 7.20. The average time comparison in performing behaviour suspension & resumption to achieve Scenario 3 – SOSJ & JADE.

Based on the results, SOSJ achieves significantly better performance than JADE in dynamic suspension and resumption, which is attributed to the fact that the JADE runtime requires more time (compared to SOSJ RTS) to exclude agents from execution and store their data and execution state (during behaviour suspension) and to retrieve agents data and execution state and include the agents for execution (during behaviour resumption). The benchmarks show that SOSJ has significantly better performance compared to JADE in handling behaviour suspension and behaviour resumption.

### 7.4.4.3 Behaviour Migration

To achieve Scenario 5, behaviour migration is performed in both SOSJ and JADE. Due to the inability of JADE to provide features that allow for weak migration, the benchmark consider only strong migration in both software frameworks. The results of the benchmark is presented in Figure
Results and Discussions

7.21. Each range bar shown in Figure 7.21 spans from the average + standard deviation ($m+s$) and average – standard deviation ($m-s$). The benchmark performs behaviour migration in JADE using two different approaches to observe any differences in the framework’s performance for both cases. The first is to trigger behaviour migration for all agents to migrate after all agents have been created (indicated as “JADE (A)”), and the second is to trigger behaviour migration for each agent to migrate once it has been created (indicated as “JADE (B)”). Meanwhile, the benchmark performs behaviour migration in SOSJ by initiating a single migration (i.e. the actual data transfer) to migrate all of the CDs to the destination. The benchmark considers the origin subsystem to run on RPI2B, while the destination subsystem runs on DE1 platform.

The results show that SOSJ requires less average time to perform complete migration than JADE. This is likely due to the use of FIPA Agent Communication Language (ACL)-based message to transport agents and their codes which introduces additional overhead to the overall migration process as the ACL message processing involves particular encoding and decoding functionalities as defined by the FIPA standard [185]. In contrast, SOSJ uses TCP/IP stream-based communication to transfer data and codes to keep performance overhead minimum while ensuring the data and codes are transferred accordingly.

The results show that JADE’s performance in case of “JADE (A)” is significantly lower than “JADE (B)”, which may be caused by either (or both) of the following reasons: (1) the overall performance of JADE decreases when a high number of agents are handled by the same runtime environment instance, which consequently affects the agent migration handling and (2) the performance of the ACL message processing functionalities in JADE decreases when higher number of agents are queued for migration. Another factor which makes the migration in SOSJ advantageous compared to JADE is that SOSJ migration handles the transfer of any dependencies27 which are used by the migrating CDs, while in JADE, this feature is not intrinsic in the framework. In JADE, programmers are responsible in ensuring any dependencies required by the migrating agents are available at the destination to use (by the migrating agents) once the migration process is finished. Note that while the average reconfiguration time increases when more CDs are run on the same embedded controller, in distributed setting each controller will run only a relatively small number of software behaviours (CDs).

27 See footnote no 17
Figure 7.21. The average time comparison in performing behaviour migration to achieve Scenario 5 – SOSJ & JADE.

7.5 Concluding Remarks

This chapter presented a set of benchmarks to showcase the performance of SOSJ and compare it against some of the existing programming frameworks, i.e., WS4D JMEDS and JADE, in addressing the programming requirements of DISS. From the comparisons, it can be concluded that SOSJ is able to satisfy all of the programming requirements, while still having a smaller memory footprint compared to both frameworks, which can only address the programming requirements of DISS to limited extent. SOSJ service invocation via signals achieves comparable performance compared to WS4D JMEDS, while service invocation via channels incurs extra performance overhead as a trade-off to more reliable service invocation. Compared to JADE, SOSJ shows better performance in achieving behaviour creation, suspension, resumption, and migration. Slightly higher performance overhead in SOSJ is observed compared to JADE in achieving behaviour termination, which can be attributed to the fact that SOSJ removes CD code from the memory while JADE doesn’t remove agent code file.
Modern DISS (e.g. industrial manufacturing systems) are becoming more and more challenging for the programmers to design due to the particular requirements associated to their target applications. In typical manufacturing systems, individual machines are reactive in nature, i.e. constantly responding to their environment as well as to collaborating machines in a consistent (or deterministic) manner. The machine behaviours are completely governed by their control software, hence the expressiveness and verification ability of the used programming language/tool and the reliability of developed software are of crucial importance. Distributed software systems also need to be highly versatile in responding to fast changing requirements. Therefore, it is important to be able to integrate new software behaviours or to reconfigure existing software behaviours to achieve different physical processes. Furthermore, machines should also be aware of the status of collaborating machines and act accordingly in maintaining the overall system functionality.

To address these challenges in DISS, various software design paradigms have been developed and used. For example, the multi-agent systems (MAS) and service oriented architecture (SOA)-based approaches are proposed to describe and implement distributed systems with software entities, namely agents and services. These entities can be added to or removed from existing DISS dynamically to change overall system behaviours. However, there exist challenges in guaranteeing deterministic reactive behaviours of each software entity (i.e. agent or service) as well as of their mutual interactions. As a result, the overall system functionalities cannot be verified or guaranteed.

There are also attempts to use languages with formal models of computation (MoC) for designing static concurrent and distributed systems. For example, SystemJ has demonstrated the ability to specify correct by construction concurrent behaviours for distributed systems. SystemJ follows GALS (Globally Asynchronous Locally Synchronous) MoC and has high expressiveness in describing and composing synchronous and asynchronous concurrent behaviours. The underlying GALS MoC and formal semantics in SystemJ allow developed software entities to be
Conclusions
correct by construction and overall system behaviours to be verifiable. However, SystemJ was
designed targeting static systems and hence introducing new software entities at runtime for
adaptive behaviour was not possible.

This chapter begins with a concise summary of the major contributions described in this
thesis. In addition, this chapter will give some ideas for potential future research.

8.1 Overview

The main goal of this thesis is to propose and explore a novel programming approach to
address the challenges in the design and programming of DISS. This programming approach is
based on the integration of the SOA paradigm with a GALS MoC-based language SystemJ with
further features to enable programmers to easily handle full dynamic behaviours, i.e., dynamic
creation, suspension, resumption, termination, and migration of software behaviours.

8.2 Main Contributions

The main contributions presented in this thesis are:

1) A novel software programming paradigm achieved through the synergy of service
oriented architecture and a formal MoC-based system-level language SystemJ

Based on the existing problems/research gaps in the state of the art of design and programming
approaches for DISS and assessment done in Section 1.3, a novel software programming paradigm
is proposed. This paradigm is based on the synergy of the service oriented architecture paradigm
and a globally asynchronous locally synchronous (GALS) model of computation based language
SystemJ. The paradigm is unique compared to the state of the art by bringing the loose-coupling
features of SOA and safe, correct by construction features of SystemJ to address the existing
problems/research gaps in the design and programming approaches for DISS. The paradigm
introduces new perspectives of CD as software behaviour which offers or uses services. The
paradigm introduces SOA functionalities, consisting of Beacon, Discovery, Notify, Discovery
Reply, Advertisement, and Request for Advertisement, which allow CD to discover, advertise, and
uses services dynamically.

2) The introduction of clock domain macro-states

In SystemJ, after execution starts, CD runs continuously in every tick. The semantics of CD
in SystemJ consider only to this extent and don’t deal with the scenario of dynamic behaviours.
Thus, in creating SOSJ, the semantics of CD need to be extended to cater for scenarios of dynamic
behaviour. The extension defines CD macro-states and their transitions, which form the CD life
Conclusions

cycle. This life cycle needs to be properly defined for every scenario of dynamic behaviour handling (i.e. dynamic creation, suspension, resumption, termination, and migration of CD), which has been presented in Chapter 4 in the form of finite state machine.

3) Descriptions of the SOSJ framework and mechanisms to handle full dynamic behaviours and reconfiguration

Based on the description of the SOSJ programming paradigm and the CD macro-states, the SOSJ framework is developed by extending the original SystemJ RTS to introduce new functionalities that comply with the paradigm proposed in Chapter 3. As a part of the SOSJ framework, the global service registry is developed which stores the service descriptions of all advertised services. The global service registry transmits Beacon (informing all parties regarding its presence), Notify (upon changes in the stored service descriptions), Discovery Reply (upon receiving Discovery), and Request for Advertisement (to request service provider to refresh their Advertisement). The SOA RTS is developed and introduced as part of the SOSJ RTS which handle SOA functionalities. The SOA RTS consists of a number of Java threads, new signal classes, and data structures (as described in Chapter 5) and is responsible for transmitting Discovery and Advertisement and receiving Discovery Reply, Notify, and Request for Advertisement. The SOA RTS provides a set of function calls and dedicated SOSJ signals which permit CDs to use the SOA functionalities and perform service invocation (and trigger channel reconfiguration when necessary). The global service registry and SOA RTS (of the service provider and consumer) achieve the interaction shown in Figure 3.2. Furthermore, additional functionalities described in Chapter 6 are also introduced to allow dynamic behaviour handling which comply with the CD macro-states and its transition described in Chapter 4. These functionalities are introduced as an extension to the SOSJ RTS and referred to as the Dynamic RTS. The Dynamic RTS consist of a number of Java threads and data structures as described in Chapter 6 and handle channel reconfiguration, dynamic creation, suspension, resumption, termination, and migration of CDs. The Dynamic RTS provides a set of function calls which allow CDs to trigger dynamic creation, suspension, resumption, termination, and migration of CDs.

4) Qualitative and quantitative comparisons between SOSJ and state of the art programming approaches, analysis, and performance evaluation of the SOSJ framework.

Assessments and comparisons are carried out to show the capability of SOSJ framework. SOSJ framework is compared qualitatively against two existing state of the art programming approaches, WS4D JMEDS (a SOA-based programming approach) and JADE (a MAS-based programming approach), based on their capabilities in satisfying the design and programming
Conclusions

requirements described in Section 1.3. Based on the qualitative assessments described in Section 1.3, it is shown that SOSJ is capable of addressing all of the specified design and programming requirements of DISS compared to both WS4D JMEDS and JADE. A set of benchmarks are run to compare the operational performance of SOSJ with WS4D JMEDS and JADE. As presented in Section 7.4.4, it is shown that SOSJ is more efficient than JADE in performing dynamic creation, suspension, resumption, and migration.

8.3 Future Works

This thesis proposes a novel programming approach SOSJ amenable for designing and programming of DISS. The thesis also describes the design and implementation of SOSJ framework and showcase its capability and performance compared to state of the art programming approaches for DISS. However, the framework itself can be considered as an initial work with a lot of room for improvements and enhancements. Among these potential works are:

1) Extended SOA interaction protocols

The SOA interaction protocol used in SOSJ is presently dependant to the global service registry application (centralized service registry). It is preferable to introduce optional SOA interaction protocol (for the programmers to choose) which doesn’t rely on a centralized service registry, where the service discovery is sent directly towards service provider (instead to a dedicated service registry like the present implementation of SOSJ framework).

2) Introducing security features

The current implementation of SOSJ framework doesn’t include built-in advanced security features. Although security may not be a huge issue in isolated network (which interconnect computing machines that run SOSJ CDs) settings, it is considered essential in non-isolated ones (e.g. internet) where potentially seamless entities may access and harm DISS. Thus, introducing some level of security is considered of utmost importance for SOSJ to be used in DISS connected to non-isolated networks. For example, the network communication functionality in SOSJ RTS could be modified to use the existing Secure Socket Layer (SSL)-based communication functionality in Java library, or alternatively, include third party SSL libraries to enable more secure and encrypted data exchange.

3) Interoperability

The current implementation of SOSJ framework provides limited built-in features to allow interactions between CDs with other software functionalities of different software framework.
Conclusions

While it would be ideal to use one programming framework for an entire DISS (i.e. SOSJ), this is not always the case and in reality many other software tools are used to describe system functionalities. For example, while some parts of the DISS used in the motivating example shown in Figure 1.1 are governed by SOSJ CDs, it involves industrial robots and typical mobile robots which may use certain middlewares (e.g. ROS [187], Player [188]) to handle physical sensors or actuators. Some of the industrial stations in the motivating example may use PLCs to handle low-level physical interfacing with industrial sensors and actuators, which they typically use the IEC 61131 or 61499 approaches. In this case, it is desirable to include interfacing functions built in the SOSJ framework to allow a certain degree of interoperability with these programming frameworks in the future. One approach in introducing interoperability of SOSJ with ROS is reported in [189].

4) Extensions to hard real-time systems, time predictable execution platforms

The existing implementation of SOSJ framework is developed to run only on JVM-enabled execution platforms. However, such execution platforms are not suitable for DISS or their parts with strict timing requirements. Java processors such as JOP [190] and its variants such as TP-JOP [191] are interesting to investigate as next potential target platforms for SOSJ due to their time predictability characteristics, and therefore, more suitable for parts of DISS with strict, hard real time requirements. Thus, it is worthy to further explore and develop SOSJ framework to be compatible with these execution platforms and the requirements of hard real-time systems.

5) Built-in advanced service matching facilities

The current implementation of SOSJ framework leaves the duty of deciding on how service matching is implemented to the programmers. This gives the programmers ample flexibility to implement service matching depending on the target applications and the computational capability of the computing machines used, although this also means additional works for the programmers. Thus, there are opportunities to include built-in service matching functions in SOSJ. For example, by including ontology (e.g. described using OWL [76]) in service description to allow ontology-based service matching.

6) Dynamic behaviour handling in SOSJ for non-round robin CD scheduling policy

The current implementation of SOSJ framework use the round robin CD scheduling policy which is used by default in the original SystemJ RTS. However, SystemJ RTS allows using other CD scheduling policies. Future development of SOSJ framework could attempt to enable dynamic behaviour handling function in case of non-round robin CD scheduling policies.
Conclusions

7) Dynamic behaviour handling in SOSJ for CDs distributed in multi-core platforms

SOSJ framework is developed initially from the original SystemJ RTS which assumes the distribution of multiple CDs in the same core in case of multi-core execution platforms. However, theoretically SystemJ RTS permits the possibility of multiple CDs in one or more cores in multi-core execution platforms. Thus, future development of SOSJ framework could be directed to include the dynamic behaviour handling features in case of multiple CDs distributed in one or more cores in multi-core execution platforms.

8) Compatibility with computationally constrained execution platforms

The current implementation of SOSJ framework is mainly aimed to target relatively powerful execution platforms which are capable to run Java Platform Standard Edition28. However, the DISS depicted in Figure 1.1 may use comparably less computationally-capable execution platforms which may not be capable of running Java Platform Standard Edition. For example, the stationary nodes of S8 and S10 in Figure 1.1 perform ambient sensing, in which they may use battery-powered wireless sensor nodes with much less power computational power to minimize power consumption and extend battery life. Future development of SOSJ framework may be aimed to address its compatibility with other types of JVMs (which may be run by much less computationally-capable execution platforms). Some execution platforms in the form of smartphones are capable to run JVMs which are also slightly different than the typical Java Platform Standard Edition. Future development of SOSJ may also consider to include compatibility with these types of platforms.

9) Enabling ‘one-to-many’ and ‘many-to-one’ SOSJ channel communication

SOSJ framework attempts to approach ‘one-to-many’ and ‘many-to-one’ interaction (in particular, for service invocation) via channel (to some extent) by introducing the mechanisms that allow channel pairing to be changed/reconfigured and paired dynamically. This approach retains the point-to-point SystemJ channel communication semantics which offer guaranteed data delivery and support the reliability of service invocation, however at the cost of quite significant additional performance overhead. Future improvement of SOSJ may include defining new channel communication semantics which permit ‘one-to-many’ and ‘many-to-one’ channel communication while also retaining the feature of rendezvous (that guarantees data delivery). The

28 Relatively powerful execution platforms include but not limited to desktop PC, laptops, and nowadays, ARM-based single board computers such as Beaglebone Black and Raspberry Pi which, albeit capable to run Java Standard Edition, their physical dimensions are much smaller compared to desktop PC and laptops.
Conclusions

improvement aims to remove the need for channel reconfiguration process and reduces service invocation performance overhead (that comes due to channel reconfiguration process).
References


[75] T. Konnerth, B. Hirsch, and S. Albayrak, "JADL – An Agent Description Language for Smart Agents," in *Declarative Agent Languages and Technologies IV: 4th International*


The Original SystemJ RTS

SOSJ RTS is developed and extended from the existing SystemJ RTS. Hence it is important to elaborate the existing SystemJ RTS to make sense of the description of the overall SOSJ RTS. Before going into more detail of the existing SystemJ RTS, it’s also important to describe how the original structure CD and subsystem configuration may also include specification of links to communicate with other subsystems.

A.1 CD and Subsystem configuration in the original SystemJ RTS with links

In the context of SystemJ where all subsystems and their interactions are specified statically and remain fixed during runtime, SystemJ requires programmers to define the configuration of links which will be used for subsystems in a SystemJ program to communicate with each other. To show how the CD and subsystem configuration which includes the configuration of links in SystemJ is structured, shows an example of a CD and subsystem configuration in SystemJ which include the configuration of links.

Listing A.1. Example of CD and subsystem configuration in SystemJ showing configuration of links

```xml
1 <System>
2  <Interconnection>
3   <Link>
4     <Interface SubSystem="SS1" Class="TCPIPInterface" Args="127.0.0.1:1112"/>
5     <Interface SubSystem="SS2" Class="TCPIPInterface" Args="127.0.0.1:1113"/>
6   </Link>
7  </Interconnection>
8  <SubSystem Name="SS1" Local="true">
9    <ClockDomain Name="CD1" Class="CD1Class">
10       <SignalChannel>
11          <iChannel Name="Ch1" From="CD2.Ch1" />
12          <oChannel Name="Ch2" To="CD2.Ch2" />
13       </SignalChannel>
14    </ClockDomain>
15  </SubSystem>
```

This means in SystemJ and SOSJ, programmers can specify the interaction between the corresponding subsystem and (some) other subsystems via links by defining the configuration of links prior to runtime.
The example in Listing A.1 shows the configuration of links which are encapsulated within the <Interconnection> tags (line 3-6). Each link configuration is specified within the Interface element (line 4-5). The SubSystem attribute defines the subsystem name. The Class attribute defines the class file which describes how the link is implemented. The example uses the TCPIInterface class which implement link as TCP/IP physical interface. The example shows two interacting subsystems, SS1 and SS2 which use two links implemented as two TCP/IP ports (via 127.0.0.1 localhost with port number 1112 and 1113, defined in the Args attributes) to allow bi-directional communications between the two subsystems. To allow the channel routing function of the InterfaceManager in the RTS (explained in Appendix A.2.9) to properly route data from one channel towards its partner channel and vice versa, the CD where the partner channel resides and also the subsystems where the CD belongs to are also described in the configuration (line 16 – 18).

**A.2 Classes in the Original SystemJ RTS**

The diagram which illustrates the Java classes of the existing SystemJ RTS and their relationships is shown in Figure A.1. As shown in Figure A.1, there are 19 Java classes in the existing SystemJ RTS, namely SystemJRunner, SystemJProgram, JDOMParser, Signal, ClockDomainAbst, GenericSignalSender, GenericSignalReceiver, BaseInterface, InterfaceManager, LinkQueue, InterConnection, Link, GenericChannel, input_Channel, output_Channel, SchedulerAbst, CyclicScheduler, GenericInterface, and TCPIPIfface. These Java classes will be elaborated in the following.
A.2.1 SystemJRunner

SystemJRunner is a Java class acting as a bootstrapping functionality in the SystemJ RTS, which is the initial functionality in the SystemJ RTS to be executed when a SystemJ subsystem begins its execution\(^{30}\). A code snippet showing the most significant code part of the SystemJRunner class is presented in Listing A.2.

Listing A.2. SystemJRunner class code snippet

```java
1            import systemj.desktop.JdomParser;
2            //... further Java package import
3
4        public class SystemJRunner() {
5            private static SystemJProgram program1;
6            private static String filename;
7            //...further variable declarations
8
9            public static void main(String[] args) {
10                parseOption(args);
11                parseXML();
12                program1.startProgram();
13            }
14            private static boolean parseOption(String[] args) {
15        }
```

\(^{30}\) This refers to the execution of all CDs in the corresponding subsystem, which is handled by the same JVM instance.
for (int i=0;i<args.length;i++) {
    if (args[i].toLowerCase().trim().endsWith(".xml")) {
        filename = args[i];
    }
}

private static void parseXML(){
    JdomParser parser;
    parser = new JdomParser(filename);
    try{
        program1 = parser.parse();
    }
    catch(Exception e){
        if(e instanceof JDOMParseException)
            System.err.println(e.getMessage());
        else
            e.printStackTrace();
    }
} //...further class descriptions

As shown in Listing A.2, the bootstrap functionality begins with the main method (line 10-14) invoking three methods of parseOption(), parseXML(), and program.startProgram() in line 11, 12, and 13. The parseOption() method (line 16-20) is executed by the SystemJRunner class to obtain the CD and subsystem configuration file name which has to be given by programmers when they initiate the execution of the subsystem. Then, the SystemJRunner invokes the parseXML() local method (line 12) which access the functionality in the JDOMParser class to parse the CD and subsystem configuration file (in XML format) provided by the programmers (line 23-35). This parsing method is invoked and passes the data obtained from the CD and subsystem configuration to “program1” (line 27), a Java object of SystemJProgram type which is capable to access and invoke the functionality provided by the SystemJProgram class to execute CDs in the subsystem. Finally, the functionality provided by the SystemJProgram to trigger the execution of CDs is invoked (line 13).

A.2.2  SystemJProgram

SystemJProgram is a Java class which invokes the execution of CDs in the subsystem based on the CD scheduling policy, the channel routing functionality provided by the InterfaceManager and the housekeeping functionality (including dynamic behaviours\reconfiguration functionality). A code snippet showing the most significant code part of the SystemJRunner class is presented in Listing A.3. As shown in Listing A.3, the SystemJProgram class has a number of methods. The setSubSystemName() and getSubSystemName() (line 8 and 9) are a pair of set and get methods to
pass the corresponding subsystem name from and to the SystemJProgram class. Similarly, the setInterfaceManager and getInterfaceManager are a pair of set and get methods to pass the information stored in and functionality offered by the InterfaceManager class to and from the SystemJProgram class (line 10 and line 11).

Listing A.3. SystemJProgram class code snippet

```java
import systemj.common.InterfaceManager;
import systemj.interfaces.Scheduler;
//...further Java package import

public class SystemJProgram {
    private String name;
    private InterfaceManager im;
    public void setSubSystemName(String n){name = n;}
    public String getSubSystemName(){ return name ;}
    public void setInterfaceManager(InterfaceManager iim){ im = iim; }
    public InterfaceManager getInterfaceManager(){ return im ;}
    private Vector scs = new Vector();
    public void addScheduler(Scheduler sc) {
        scs.addElement(sc);
    }
    public void init(){
        im.init();
        for(int i=0;i<scs.size();i++)
            ((Scheduler)scs.elementAt(i)).setInterfaceManager(im);
    }
    public void startProgram(){
        System.out.println("Starting program");
        while(true){
            for(int i=0;i<scs.size();i++)
                ((Scheduler)scs.elementAt(i)).run();
            im.run();
        }
    }
    public Scheduler getScheduler (){ return (Scheduler)scs.get(0);}
}
```

The addScheduler() method is used to initialise a CD scheduler instance which will be used by the RTS to scheduler the execution of CDs in the subsystem (line 14-16). The init() method is a method used to initialise the channel routing function in the scheduler function (line 18-22). The startProgram() method triggers the CD scheduling policy which execute the CDs in the

---

Note that while the existing RTS implementation allows more than one CD scheduling policies, this thesis assumes the use of one CD scheduling policy and the round-robin (cyclic) scheduling policy.

---

31 Note that while the existing RTS implementation allows more than one CD scheduling policies, this thesis assumes the use of one CD scheduling policy and the round-robin (cyclic) scheduling policy.
subsystem (line 28) and the channel routing function (line 29). The getScheduler method is used to obtain the scheduler instance (line 32).

### A.2.3 JDOMParser

The JDOMParser class provides the functionality to parse the CD and subsystem configuration file which is written in XML format and instantiate the subsystem. The class also includes the functionality to initialise CD descriptors, signals, channels, and links (if any is defined by programmers during design time) based on the provided CD and subsystem configuration file. Note that this class uses the JDOM version 1.0 library which can be used to store information from XML document in Java [192].

Listing A.4. JDOMParser class code snippet

```java
1 //...Java import...//
2
3 public class JdomParser {
4    private String file;
5    private String SSName;
6    private SystemJProgram program;
7    private int gid = 0;
8    public JdomParser(String file){
9        this.file = file;
10        program = new SystemJProgram();
11    }
12
13    public SystemJProgram parse() throws Exception{
14        SAXBuilder builder = new SAXBuilder();
15        Document doc;
16        File f = new File(file);
17        doc = builder.build(f);
18        Element e = doc.getRootElement();
19        parseSubSystem(e);
20        return program;
21    }
22
23    private void parseSubSystem(Element el){
24        List<Element> subsystems = el.getChildren("SubSystem");
25        List<Element> intercon = el.getChildren("Interconnection");
26        InterfaceManager im = new InterfaceManager();
27        parseInterconnection(intercon.get(0), im);
28        Element ss = makeMap(subsystems, im);
29        parseAllCDs(ss, im);
30        program.setInterfaceManager(im);
31        program.init();
32    }
33
34    public Element makeMap(List<Element> el, InterfaceManager im){
35        Element localsub = null;
36        try {
37            for(Element subsystem : el){
38                String name = subsystem.getAttributeValue("Name");
39                localsub = subsystem;
40                constructMap(subsystem, name, im);
41            }
42        }
43    }
```
try {
    cdins = (ClockDomain) Class.forName(clazz).newInstance();
    } catch (Exception ee) {
        ee.printStackTrace();
    }
    cdins.setName(name);
    clockdomains.put(name, cdins);
}
List<Element> l = e.getChildren();
for (Element ee : l)
    this.createCDInstances(ee, level, clockdomains);

return localsub;

public void constructMap(Element el, String ssname, InterfaceManager im) {
    List<Element> l = el.getChildren();
    for (Element ee : l)
        constructMap(ee, ssname, im);
}

public void parseAllCDs(Element subsystem, InterfaceManager im) {
    List<Element> cds = subsystem.getChildren("ClockDomain");
    program.setSubSystemName(subsystem.getAttributeValue("Name"));
    im.setLocalInterface(subsystem.getAttributeValue("Name"));
    Hashtable clockdomains = new Hashtable();
    createCDInstances(subsystem, 0, clockdomains);
    Hashtable channels = new Hashtable();
    if(cds.size() > 0){
        CyclicScheduler cs = new CyclicScheduler();
        for(Element cd : cds){
            ClockDomain cdd = parseClockDomain(cd, subsystem.getAttributeValue("Name"), channels, im, clockdomains);
            cs.addClockDomain(cdd);
        }
        program.addScheduler(cs);
        im.setChannelInstances(channels);
    }

    private void createCDInstances(Element e, int level, Hashtable clockdomains) {
        if(e.getAttributeValue("Name") != null){
            String name = e.getAttributeValue("Name");
            String clazz = e.getAttributeValue("Class");
            ClockDomain cdins = null;
            try {
                cdins = (ClockDomain) Class.forName(clazz).newInstance();
            } catch (Exception ee){
                ee.printStackTrace();
            }
            cdins.setName(name);
            clockdomains.put(name, cdins);
        }
        List<Element> l = e.getChildren();
        for (Element el : l){
            this.createCDInstances(el, level, clockdomains);
        }
    }

    public ClockDomain parseClockDomain (Element cd, String ssname, Hashtable channels, InterfaceManager im, Hashtable clockdomains){
        String cdname = cd.getAttributeValue("Name");
        return localsub;
    }

    catch (DataConversionException e) {
        e.printStackTrace();
        System.exit(1);
    }
    return localsub;
ClockDomain cdins = (ClockDomain)clockdomains.get(cdname);
List<Element> ports = cd.getChildren();
for(Element port : ports)
    GenericSignalReceiver server = null;
    GenericSignalSender client = null;
    String portname = port.getName();
    try {
        List<Attribute> attributes = port.getAttributes();
        Hashtable config = new Hashtable();
        for(Attribute attribute : attributes){
            config.put(attribute.getName(), attribute.getValue());
        }
        if(portname.equals("iSignal")){
            server = (GenericSignalReceiver) Class.forName(port.getAttributeValue("Class")).newInstance();
            server.cdname = cdname;
            server.configure(config);
            Field f = cdins.getClass().getField(port.getAttributeValue("Name"));
            Signal input_Channel = (Signal)f.get(cdins);
            signal.setServer(server);
            signal.setuphook();
            signal.setInit();
        } else if(portname.equals("oSignal")){
            client = (GenericSignalSender) Class.forName(port.getAttributeValue("Class")).newInstance();
            client.cdname = cdname;
            client.configure(config);
            Field f = cdins.getClass().getField(port.getAttributeValue("Name"));
            Signal output_Channel = (Signal)f.get(cdins);
            signal.setClient(client);
            signal.setInit();
        } else if(portname.equals("iChannel"))
            String cname = port.getAttributeValue("Name").trim() + "+_in";
            String pname = port.getAttributeValue("From").trim() + "+_o";
        if(!channels.containsKey(cdname + "." + cname))
            if(SSName.equals(im.getCDColocation(pnames[0])))
                Field f = cdins.getClass().getField(cname);
                Field f2 = partnercd.getClass().getField(pnames[1]);
                input_Channel inchan = (input_Channel)f.get(cdins);
                output_Channel ochan = (output_Channel)f2.get(partnercd);
                inchan.setInit();
                ochan.setInit();
                inchan.Name = cdname + "." + cname;
                inchan.PartnerName = pname;
                ochan.Name = ochan;
                ochan.PartnerName = cdname + "." + cname;
                inchan.set_partner_smp(ochan);
                ochan.set_partner_smp(inchan);
                channels.put(inchan.Name, inchan);
                channels.put(ochan.Name, ochan);
            } else {
                Field f = cdins.getClass().getField(cname);
                input_Channel inchan = (input_Channel)f.get(cdins);
                inchan.setInit();
                inchan.Name = cdname + "." + cname;
inchanPartnerName = pname;
inchan.setDistributed();
inchan.setInterfaceManager(im);
}
}

else if(portname.equals("oChannel")) {
    String cname = port.getAttributeValue("Name").trim() + "_o";
    String pname = port.getAttributeValue("To").trim() + "_in";
    String[] pnames = pname.split(/\./);
    if(!channels.containsKey(cdname + "+" + cname)) {
        if(SSName.equals(im.getCDLocation(pnames[0]))) {
            ClockDomain partnercd = (ClockDomain) clockdomains.get(pnames[0]);
            Field f = cdins.getClass().getField(cname);
            Field f2 = partnercd.getClass().getField(pnames[1]);
            output_Channel ochan = (output_Channel)f.get(cdins);
            input_Channel inchan = (input_Channel)f2.get(partnercd);
            ochan.setInit();
            inchan.setInit();
            ochan.Name = cdname + "+" + cname;
            ochan.PartnerName = pname;
            inchan.Name = pname;
            inchan.PartnerName = cdname + "+" + cname;
            inchan.set_partner_smp(ochan);
            ochan.set_partner_smp(inchan);
            if(!channels.containsKey(inchan.Name)) {
                channels.put(inchan.Name, inchan);
                channels.put(ochan.Name, ochan);
            }
        } else { 
            Field f = cdins.getClass().getField(cname);
            output_Channel ochan = (output_Channel)f.get(cdins);
            ochan.setInit();
            ochan.Name = cdname + "+" + cname;
            ochan.PartnerName = pname;
            ochan.setDistributed();
            ochan.setInterfaceManager(im);
            if(!channels.containsKey(ochan.Name)) {
                channels.put(ochan.Name, ochan);
            }
        }
    } else {
        Field f = cdins.getClass().getField(cname);
        output_Channel ochan = (output_Channel)f.get(cdins);
        ochan.setInit();
        ochan.Name = cdname + "+" + cname;
        ochan.PartnerName = pname;
        ochan.setDistributed();
        ochan.setInterfaceManager(im);
        if(!channels.containsKey(ochan.Name)) {
            channels.put(ochan.Name, ochan);
        }
    }
} catch(Exception e) {
    e.printStackTrace();
}

return cdins;
}

public void parseInterconnection(Element el, InterfaceManager im) {
    List<Element> l = el.getChildren("Link");
    Interconnection ic = new Interconnection();
    for(Element link : l) {
        Interconnection.Link linko = new Interconnection.Link();
        List<Element> intfS = link.getChildren("Interface");
        for(Element intf : intfS) {
            String SS = intf.getAttributeValue("SubSystem");
String clazz = intf.getAttributeValue("Class");
String args = intf.getAttributeValue("Args");
try {
    GenericInterface gct = (GenericInterface)Class.forName(clazz).newInstance();
    Hashtable ht = new Hashtable();
    ht.put("Class", clazz);
    ht.put("Args", args);
    ht.put("SubSystem", SS);
    gct.configure(ht);
    linko.addInterface(SS, gct);
} catch(Exception e) {
    e.printStackTrace();
}
ic.addLink(linko);
im.setInterconnection(ic);

The parsing function is initiated when the parse() method is invoked by the SystemJRunner class in line 27 of Listing A.2. As shown in Listing A.4, the parse() method is described in line 12-20. When the parse() method is invoked, the method obtains the XML file to be parsed, which is passed to the JDOMParser class through the constructor method (line 9-10), and converts the information in the file as a representation in JDOM of Element type (line 13-17). Then, the parse() method invokes the parseSubSystem() method (line 18). The parseSubSystem() method obtains the information regarding the name of CDs belonging to other subsystems and the corresponding subsystems they belong to (if specified by programmers during design time) by instantiating the channel routing function InterfaceManager, parsing the configuration of the link specified by the programmers in the configuration file and instantiating links based on the configuration by invoking the parseInterconnection() method, and embedding the link configuration to the channel routing function in the InterfaceManager for inter-subsystems data exchange purposes via channels (line 24-26). Then, the parseInterconnection method (which is described in line 204-229) is invoked to initial links based on the provided configuration³². Once links are instantiated (if any is specified and instantiated), the MakeMap() method (line 26) is invoked which then it invokes the constructMap() method (line 38) to obtain the subsystem name included in the configuration and the name of the CDs belonging to the subsystem required by the channel routing function of InterfaceManager. Then, the parseAllCDs() method is invoked to parse the configuration of CD and its signals and channels (line 27). The parseAllCDs() method recursively goes to the configuration of individual CDs to instantiate each CD and its signals and channels (line 56, line

³² The existing RTS implements links as TCP/IP physical ports. This thesis follows this assumption to constraint the implementation of links as TCP/IP physical ports.
and extracts the name of the corresponding subsystem and loads it to the channel routing function for the channel routing function to know the name of the corresponding subsystem (line 57). The parseClockDomain() method is invoked to parse the configuration of individual CDs and their signals and channels (line 66). The method instantiates individual CDs signals and channels (line 94-203).

In regards to the instantiation of signals and channels, the input signal instantiation functionality is described in line 108-117. The instantiation of each input signal is accompanied with the instantiation of GenericSignalReceiver which buffers the information captured by the physical interface implementation of the input signal to the input signal itself (line 109-111). The input signal is then instantiated, along with its corresponding GenericSignalReceiver and the implementation of the physical interface used by the input signal (line 112-116). The output signal instantiation functionality is described in line 118-126. In contrast to the output signal, the instantiation of each output signal is accompanied with the instantiation of GenericSignalSender which buffers the data transferred by the output signal to the physical interface implementation of the output signal (line 118-125). The instantiation of input channel is described in line 127-159. Initially, the configuration of the corresponding input channel is obtained, i.e. the name and the partner channel’s name, and then new String representation of both configuration are created to be able to locate the Java Field associated with the input channel and its partner channel from the CD descriptors of the corresponding CD and the partner CD, respectively (line 128-129). The input channel and its partner channel objects are located through the aforementioned Java Field, which then they are configured and instantiated based on the provided configuration (line 134-158). The input channel can be used to either communicate with a partner CD which belongs to the same subsystem, or a partner CD which belongs to a different subsystem. If the partner CD belongs to the same subsystem, the functionality which configures the input channel and the partner channel is described in line 132-147 are executed, otherwise, the functionality described in line 149-157 are executed. The instantiation of output channel is shown in line 160-197. Initially, the configuration of the corresponding output channel is obtained, i.e. the name and the partner channel’s name, and then new String representation of both configuration are created to be able to locate the Java Field associated with the output channel and its partner channel from the CD descriptors of the corresponding CD and the partner CD, respectively (line 161-162). Then the output channel and its partner channel objects are located through the aforementioned Java Field, which then they are configured and instantiated based on the provided configuration (line 167-182). Similar to the input channel, the output channel can be used to either communicate with a partner CD which belongs to the same subsystem, or a partner CD which belongs to a different
subsystem. If the partner CD belongs to the same subsystem, the functionality which configures the output channel and the partner channel (line 132-147) are executed, otherwise, the functionality described in line 149-157 are executed. The description of the functionalities used to configure input and output channels shows that the coupling between both input and output channels are determined by the Name and PartnerName variables of both the input and output channel objects of the corresponding input and output channel pair, respectively.

Once signals and channels are instantiated, all CDs in the subsystem are instantiated by the invocation of createCDInstances() method (line 61). Finally, after all CDs in the subsystem are instantiated, they are included in the scheduler for execution (line 67).

A.2.4 Signal

The Signal class defines the how SystemJ signals are implemented in Java. A code snippet showing the Signal class implementation is shown in Listing A.5. Note that the name Signal is used to differentiate the Java class implementation of the signal from signal as abstracted communication mechanism used by SystemJ. The implementation of this class is described in Listing A.5.

Listing A.5. Signal class code snippet

```java
import java.util.*;
import systemj.bootstrap.SystemJProgram;
import systemj.interfaces.*;

public class Signal {

private boolean status = false;
public Object value = null;
public Object pre_val = null;
public boolean pre_status = false;
private Signal partner = null;
private GenericSignalReceiver server;
private GenericSignalSender client;
private Object[] toSend = new Object[2];
private Object[] toReceive = new Object[2];
private boolean init = false;

public Signal(){
}
public boolean isInit(){ return init; }
public void setInit(){ init = true; }
public void setServer(GenericSignalReceiver gsr){ server = gsr; }
public void setClient(GenericSignalSender gss){ client = gss; }
public void setuphook(){
    try{new Thread(server).start();}
    catch(Exception e){e.printStackTrace();}
}
public void gethook(){
    if(server != null){
        server.getBuffer(toReceive);
    }
}
```
if(((Boolean)toReceive[0]).booleanValue()){
    this.status = true;
    this.value = toReceive[1];
} else
    this.status = false;

public void sethook(){
    if(client != null){
        toSend[0] = Boolean.TRUE;
        toSend[1] = value;
        if(client.setup(toSend))
            client.run();
    }
}

...further class description...//

The class contains several methods. The setInit() and getInit() methods (line 18-19) are get and set methods which access an internal flag that indicates whether a particular signal has been configured and ready to use. The setServer() and setClient() methods (line 20-21) defines the GenericSignalReceiver and GenericSignalSender objects to be associated to the signal, respectively. The setuphook() method (line 22-25) initiates any Java thread implementations of physical interface used by the signals. The gethook() method (line 26-36) fetches any data from the physical interface implementation used by input signal and passes it to the signal object (if the signal is an input signal) to be read by the CD. The sethook() method (line 37-44) buffers data sent by output signal and passes it to the physical interface implementation used by the output signal.

A.2.5 ClockDomain

The ClockDomain class is a Java abstract class used by CD code class file. When CD code written in SystemJ is compiled to Java, the compilation process generates Java class of individual CD that inherits the ClockDomain class. A code snippet showing the ClockDomain class is shown in Listing A.6. The class defines two methods, getName() and setName(), in which both are get and set methods used to store the name of the CD. The init() method is available to contain the initialization function of the CD after it begins its execution. The runClockDomain() method contains the description of the CD behaviour as translated into Java through compilation process.

Listing A.6. ClockDomain class code snippet.
A.2.6 GenericSignalSender

The GenericSignalSender class is a Java abstract class which is used to implement buffering functionality between output signal and the physical interface implementation used by the output signal. A code snippet showing the GenericSignalSender class is shown in Listing A.7

Listing A.7. GenericSignalSender class code snippet.

```java
import java.util.Hashtable;

public abstract class GenericSignalSender implements Runnable {
    public String name;
    public String cdname;
    protected Object[] buffer = new Object[2];
    public boolean setup(Object[] obj){
        if(buffer.length < obj.length)
            buffer = new Object[obj.length];
        for(int i=0; i<obj.length; i++)
            buffer[i] = obj[i];
        return true;
    }
    public abstract void configure(Hashtable data) throws RuntimeException;
    public abstract void run();
}
```

As shown in Listing A.7, the GenericSignalSender class has three methods. The setup() method (line 8-14) copies the content of the ‘buffer’ Object array which contains the signal status and value (if any) if the output signal is emitted. The configure() method (line 15) contains the initialisation function for the physical interface implementation of the output signal. The run() method (line 16) contains the description on how the physical interface of the output signal is implemented.

A.2.7 GenericSignalReceiver

The GenericSignalReceiver class is a Java abstract class which is used to implement buffering functionality between input signal and the physical interface implementation used by the input signal. A code snippet showing the GenericSignalReceiver class is shown in Listing A.8.

```java
import java.util.Hashtable;

public abstract class GenericSignalReceiver implements Runnable {
    public String name;
    public String cdname;
    protected Object[] buffer = new Object[2];
    public GenericSignalReceiver() {
        buffer[0] = Boolean.FALSE;
    }
    public synchronized void getBuffer(Object[] obj) {
        for(int i = 0; i < buffer.length; i++)
            obj[i] = buffer[i];
        buffer[0] = Boolean.FALSE;
    }
    public synchronized void setBuffer(Object[] obj) {
        if(buffer.length < obj.length)
            buffer = new Object[obj.length];
        for(int i = 0; i < obj.length; i++)
            buffer[i] = obj[i];
    }
    public abstract void configure(Hashtable data) throws RuntimeException;
    public abstract void run();
}
```

Referring to Listing A.8, the GenericSignalReceiver class has four methods. The getBuffer() method (line 10-14) obtains the content of the ‘buffer’ Object array (which contains data passed by the physical interface implementation used by the input signal from the environment). This method can also be extended to contain how the physical interface used by the input signal is implemented. The setBuffer() method (line 15-20) stores data from the physical interface implementation captured from the environment. The configure() method (line 21) initialises the physical interface implementation used by the input signal when the input signal is instantiated. The run() method (line 22) can be used to describe how the physical interface used by the input signal is implemented.

### A.2.8 BaseInterface

The BaseInterface class is a Java abstract class in the SystemJ RTS. It is used by the InterfaceManager, Scheduler, and GenericInterface class. The code snippet of this class is shown in Listing A.9.

Listing A.9. The BaseInterface class code snippet

```java
public class BaseInterface {
    private InterfaceManager im;
    public final void setInterfaceManager(InterfaceManager imm) { im = imm; }
    public final boolean pushToQueue(Object[] o) { return im.pushToQueue(o); }
    public final void runInterfaceManager() { im.run(); }
}
```
Referring to Listing A.9, the BaseInterface class defines four methods. The setInterfaceManager() method (line 3) initialises InterfaceManager object instance. The pushToQueue() method (line 4) passes data to the InterfaceManager by invoking the pushToQueue() method of the InterfaceManager. The runInterfaceManager() method (line 5) invokes the run() method of the InterfaceManager() to initiate channel routing function. The forwardChannelData() method (line 6) invokes the forwardChannelData() of the InterfaceManager. More details regarding the functionality in the InterfaceManager class will be explained in Appendix A.2.9

A.2.9 InterfaceManager

The InterfaceManager class is Java class which contains channel routing function. The code snippet of this class is shown in Listing A.10.

Listing A.10. The InterfaceManager class code snippet.

```java
import systemj.common.util.LinkQueue;
import systemj.interfaces.GenericChannel;
import systemj.interfaces.GenericInterface;

public class InterfaceManager {
    private String ssname;
    private LinkQueue OutQueue = new LinkQueue();
    private Vector LocalInterface = new Vector();
    private Interconnection ic = new Interconnection();
    private Hashtable cdlocation = new Hashtable();
    private Hashtable chanins = new Hashtable();
    private Hashtable cachedintf = new Hashtable();
    private Vector unsentdata = new Vector();
    private final int MAX_UNSENT_DATA = 50;

    public void addCDLocation(String ss, String cd){
        cdlocation.put(cd, ss);
    }

    public String getCDLocation(String cd){
        return (String)cdlocation.get(cd);
    }

    public void setChannelInstances(Hashtable ci){
        chanins = ci;
    }

    private Object getChannelInstances(String n){
        return chanins.get(n);
    }

    public void setInterconnection(Interconnection ic){
        this.ic = ic;
    }

    public void setLocalInterface(String ssname){
        LocalInterface = ic.getInterfaces(ssname);
        this.ssname = ssname;
    }

    public final void forwardChannelData(Object[] o){
        im.forwardChannelData(o);
    }
}
```
public void init()
for(int i=0;i<LocalInterface.size(); i++){
    ((GenericInterface)LocalInterface.elementAt(i)).invokeReceivingThread();
    ((GenericInterface)LocalInterface.elementAt(i)).setInterfaceManager(this);
}

public boolean pushToQueue(Object o){
    if(OutQueue.isFull())
        return false;
    else
        return OutQueue.push(o);
}

private synchronized void addToUnsent(Object[] o){
    boolean there =false;
    for(int i=0;i<unsentdata.size(); i++)
        if(((Object[])unsentdata.elementAt(i))[0].equals(o[0])){
            unsentdata.setElementAt(o, i);
            return;
        }
    if(unsentdata.size() < MAX_UNSENT_DATA)
        unsentdata.addElement(o);
}

private boolean tryToSend(Object[] o){
    String destcd = ((String)o[0]).substring(0, ((String)o[0]).indexOf(".")));
    String dest = this.getCDLocation(destcd);
    GenericInterface gi = null;
    if(cachedintf.containsKey(dest)){
        gi = (GenericInterface)cachedintf.get(dest);
        gi.setup(o);
    }
    else{
        Vector l = ic.getInterfaces(ssname, dest);
        if(l.size() > 0){
            cachedintf.put(dest, l.elementAt(0));
            gi = (GenericInterface).elementAt(0);
            gi.setup(o);
        }
        else{
            return false;
        }
    }
    if(!gi.transmitData()){
        Vector l = ic.getInterfaces(ssname, dest);
        for(int i=0;i<l.size(); i++)
            gi = (GenericInterface).elementAt(i);
        gi.setup(o);
        if(gi.transmitData()){
            cachedintf.put(dest, gi);
            return true;
        }
    }
    return false;
}

else
    return true;
public void forwardChannelData(Object[] o){
    GenericChannel chan = (GenericChannel)getChannelInstance(((String)o[0]));
    String destcd = ((String)o[0]).substring(0, ((String)o[0]).indexOf("."));
    if(chan == null){
        if(this.getCDLocation(destcd).equals(ssname)){
            this.addToUnsent(o);
        }
    } else
        chan.setBuffer(o);
}

private synchronized void resendUnsent(){
    for(int i=0;i<unsentdata.size(); i++){
        boolean done = tryToSend((Object[])unsentdata.elementAt(i));
        if(done){
            unsentdata.removeElementAt(i);
            i--;
        }
    }
}

private void transmit(){
    if(unsentdata.size() > 0)
        resendUnsent();
    while(!OutQueue.isEmpty()){
        Object[] o = (Object[])OutQueue.pop();
        boolean done = tryToSend(o);
        if(!done)
            this.addToUnsent(o);
    }
}

public void run(){
    transmit();
}

//…further class description…//

Referring to Listing A.10, the InterfaceManager class has a number of methods. The addCDLocation() and getCDLocation() methods are set and get methods used to store and obtain the information of CDs and the subsystems they belong to, respectively (line 16-21). The setChannelInstances() and getChannelInstances() are also set and get methods used to store and obtain channel objects and their pairing information for the purpose of channel routing function by the InterfaceManager class (line 22-27). The setInterconnection() method (line 28-30) stores created link objects to the InterfaceManager for the purpose of channel routing function to be able to transmit and receive channel data via the proper links (in the case of channel communication between CDs belonging to different subsystems). The init() method (line 35-40) initialises the execution of receiving function of links. The pushToQueue() method (line 41-46) invokes the push() method in LinkQueue class to buffer data before for transmission via links. The addToUnsent() method (line 47-57) stores data to be transmitted in the next attempt if the
transmission fails at one attempt. The tryToSend() method (line 58-91) attempts to transmit channel data via links and handle its routing. The forwardChannelData() method (line 92-102) passes data which is received via links to the channel objects accordingly. The resendUnsent() method (line 103-111) provides the mechanism to re-attempt channel data transmission to the destination. The transmit() method (line 112-121) triggers the invocation of tryToSend() method. The run() method invokes the transmit() method (line 122-124).

A.2.10 LinkQueue

The LinkQueue class is a data structure class used by the InterfaceManager to temporarily store channel data. The code snippet describing the LinkQueue class is shown in Listing A.11.

Listing A.11. LinkQueue class code snippet.

```java
public class LinkQueue {
    private Vector q = new Vector();
    private static final int MAX_SIZE = 50;
    private int currentsize = 0;

    public synchronized boolean push(Object o){
        if(q.size() < MAX_SIZE){
            q.addElement(o);
            currentsize = q.size();
            return true;
        }
        currentsize = q.size();
        return false;
    }

    public synchronized Object pop(){
        Object o = null;
        if(q.size() > 0){
            o = q.elementAt(0);
            q.removeElementAt(0);
        }
        currentsize = q.size();
        return o;
    }

    public boolean isFull(){ return currentsize > MAX_SIZE; }
    public boolean isEmpty(){ return currentsize == 0; }
}
```

Referring to Listing A.11, the LinkQueue class has four methods. The push() and pop() methods (line 6-23) stores and accesses data in the data structure in the LinkQueue class. The isFull() and isEmpty() methods (line 24-25) checks the data structure whether it is full or empty in the LinkQueue class.
A.2.11 Interconnection and Link

The Interconnection class stores link objects and the association of individual links with the subsystems the corresponding links are used to communicate with. The code snippet describing the Interconnection class is shown in Listing A.12.

Listing A.12. The code snippet describing the Interconnection class.

```java
import systemj.interfaces.GenericInterface;

public class Interconnection {
    private Vector DestLinks = new Vector();

    public static class Link {
        public Hashtable InterfaceMap = new Hashtable();
        public Vector Keys = new Vector();

        public void addInterface(String SS, GenericInterface gct) {
            Keys.addElement(SS);
            InterfaceMap.put(SS, gct);
        }

        public void addLink(Link link) {
            DestLinks.addElement(link);
        }

        public Vector getInterfaces(String ssname) {
            Vector l = new Vector();
            for (int i = 0; i < DestLinks.size(); i++) {
                GenericInterface gct = (GenericInterface)((Link)DestLinks.elementAt(i)).InterfaceMap.get(ssname);
                if (gct != null)
                    l.addElement(gct);
            }
            return l;
        }
    }
}
```

Referring to Listing A.12, the Interconnection class has three methods and the Link class. The Link class contains link and the association of links with the subsystem the links are used to communicate with. The addInterface() method (line 10-13) stores the created link and their association to the subsystem the links are used to communicate with. The addLink() method (line 15-17) stores Link objects. The getInterfaces() method (line 19-27) obtains stored Link objects.

A.2.12 GenericChannel

The GenericChannel class is a Java abstract class which inherits the BaseInterface class and is used by input_Channel and output_Channel class. The code snippet describing the GenericChannel class is shown in Listing A.13. Referring to Listing A.13, the GenericChannel class has three methods. The setInit() method (line 15) is used to set the flag in the GenericChannel.
class that channel has been initialised. The isInit() method (line 14) is used to check the flag in the 
GenericChannel whether the channel has been initialised. The setBuffer() method (line 16-22) is 
used to pass channel data to be stored in the channel (input or output) objects, which will inherit 
this class and can access the channel data.

Listing A.13. The code snippet describing the GenericChannel.

```java
import systemj.common.BaseInterface;

public abstract class GenericChannel extends BaseInterface{

    protected boolean modified = false;
    protected boolean init = false;
    protected Object value;
    public String PartnerName;
    public String Name;
    protected boolean isLocal = true;
    protected Object[] toReceive = new Object[5];
    protected boolean incoming = false;

    public GenericChannel() {}

    public boolean isInit(){ return init; }

    public void setInit(){ init = true; }

    public synchronized void setBuffer(Object[] obj){
        if(toReceive.length < obj.length)
            toReceive = new Object[obj.length];
        for(int i=0;i<obj.length; i++)
            toReceive[i] = obj[i];
        incoming = true;
    }
}
```

**A.2.13 input_Channel**

The input_Channel class is a Java class which describes the implementation of input channel 
in Java. The class inherits the GenericChannel class and is able to access channel data stored in 
the GenericChannel class. The code snippet describing the input_Channel class is shown in Listing 
A.14.

Listing A.14. The input_Channel class code snippet.

```java
import systemj.interfaces.*;

public class input_Channel extends GenericChannel{

    private int preempted = 0;
    private int r_r = 0;
    private int r_s = 0;
    public output_Channel partner;

    public input_Channel()
    {
        this.partner = new output_Channel();
    }

    public void set_partner(output_Channel partner)
    {
        this.partner = partner;
    }

    public Object get_value()
    {
        return this.value;
    }
```

233
public void set_r_r(int in){this.r_r = this.r_s; this.modified = true;}
public int get_r_r(){return this.r_r; }
public int get_r_s(){return this.r_s; }
private int get_w_s(){
    return init ? partner.get_w_s() : 0;
}
public void update_r_s(){
    if(init){
        if(partner.get_preempted_val() == this.preempted)
            this.r_s = get_w_s();
    }
}
public int get_preempted_val(){return this.preempted; }
public void set_preempted() {++this.preempted; ; this.modified = true;}
public boolean get_preempted() {
    if(init){
        if(partner.get_preempted_val() > this.preempted || this.r_s < this.r_r)
            return true;
    }
    return false;
}
public void set_preempted(int num){this.preempted=num; this.modified = true;}
public void get_val(){
    if(init)
        this.value = partner.get_value();
}
public int refresh(){
    this.value = null;    this.r_r = 0;
    this.r_s = 0;
    set_preempted();
    this.modified = true;
    return 1;
}
public void gethook(){
    if(init){
        if(isLocal)
            partner_ref.updateLocalPartner(this.partner);
        else if(incoming){
            this.getBuffer();
            if(partner.get_preempted_val() < this.preempted)
                modified = true;
        }
    }
}
public synchronized void getBuffer(){
    partner.set_w_s(((Integer)toReceive[1]).intValue());
    partner.set_w_r(((Integer)toReceive[2]).intValue());
    partner.set_preempted(((Integer)toReceive[3]).intValue());
        partner.set_value(toReceive[4]);
    incoming = false;
}
public void setDistributed(){ isLocal = false; partner = new output_Channel();}
private output_Channel partner_ref = null;
private int r_r_copy = 0;
private int r_s_copy = 0;
Referring to Listing A.14, the set_partner() method (line 9) defines the output_Channel object in the input_Channel class which is used to retain a copy of the communication statuses of the partner output channel. The methods shown in line 10-39 are set and get methods of the input channel’s communication statuses and value. More details regarding both input and output channel’s communication statuses are described in detail in [61]. The refresh() method (line 40-46) resets the input channel’s communication statuses and value buffer to the initial condition. The gethook() method (line 47-57) obtains the updated partner channel’s communication statuses and value. In the case of partner channel in the same subsystem, the partner channel’s communication statuses and value are accessed directly via the partner output_Channel object and copied to the output_Channel object in the input_Channel class, while in the case of partner channel in a different subsystem, the partner channel’s communication statuses and value are obtained by invoking the getBuffer() method, which accesses the partner channel’s communication statuses.
and value stored in the buffer of the GenericChannel class. The setDistributed() method (line 66) sets a flag in the input_Channel object indicating that the input channel is used to communicate with a partner channel in a different subsystem. The sethook() method (line 71-88) stores a copy of the input channel’s communication statuses and value (if any) to local variables in input_Channel class, which are accessible by the partner channel (in the case of partner channel in the same subsystem). In the case of partner channel in a different subsystem, the sethook() method makes an attempt to transmit the input channel’s communication statuses and value to the partner channel via link. The set_partner_smp() method (line 89-92) creates a mechanism to allow the input channel to access the partner channel’s object and obtain the partner channel’s communication statuses and value (only applicable in the case of partner channel in the same subsystem). The updateLocalPartner() method (line 93-99) obtains the updated partner channel’s communication statuses and value (if any) and stores them to the output_Channel object as a copy of the partner channel’s output_Channel object. The updateLocalCopy() method (line 100-104) updates the copy of the input channel’s communication statuses with the updated input channel’s communication statuses.

A.2.14 output_Channel

The output_Channel class is a Java class which describes the implementation of output channel in Java. Similar to the input_Channel class, the class inherits the GenericChannel class and is able to access channel data stored in the GenericChannel class. The code snippet describing the output_Channel class is shown in Listing A.15.

Listing A.15. The output_Channel class code snippet.

```java
import systemj.interfaces.*;

public class output_Channel extends GenericChannel{
    public input_Channel partner;
    private int preempted = 0;
    private int w_s = 0;
    private int w_r = 0;
    public output_Channel(){
        public void set_partner(input_Channel partner){this.partner = partner;}
        public void set_value(Object in){this.value = in; this.modified = true;}
        public Object get_value(){return this.value; }
        public void set_w_s(int in){this.w_s = in; this.modified = true;}
        public void set_w_r(int in){this.w_r = in; this.modified = true;}
        public int get_w_s(){return this.w_s;}
        public int get_w_r(){return this.w_r;}
        private int get_r_r(){return init ? partner.get_r_r() : 0;}
        public void update_w_r(){
            if(init){
```

33 Only used in the case of partner channel in the same subsystem with the input channel.
if (this.preempted == partner.get_preempted_val())
    this.w_r = get_r_r();

public int get_preempted_val(){return this.preempted; }

public void set_preempted() {++this.preempted; ; this.modified = true;}

public boolean get_preempted(){
    if(init){
        if(partner.get_preempted_val() > this.preempted || this.w_s < this.w_r)
            return true;
    }
    return false;
}

public void set_preempted(int num){this.preempted=num; this.modified = true;}

public int refresh(){
    this.value = null;
    this.w_s = 0;
    this.w_r = 0;
    set_preempted();
    this.modified = true;
    return 1;
}

public void gethook(){
    if(init){
        if(isLocal)
            partner_ref.updateLocalPartner(this.partner);
        else if(incoming){
            this.getBuffer();
            if(partner.get_preempted_val() < this.preempted)
                modified = true;
        }
    }
}

public synchronized void getBuffer(){
    partner.set_r_s(((Integer)toReceive[1]).intValue());
    partner.set_r_r(((Integer)toReceive[2]).intValue());
    partner.set_preempted(((Integer)toReceive[3]).intValue());
    incoming = false;
}

public void sethook(){
    if(init & this.modified){
        if(isLocal){
            updateLocalCopy();
        }
    }
    else{
        Object[] toSend = new Object[5];
        toSend[0] = PartnerName;
        toSend[1] = new Integer(this.get_w_s());
        toSend[2] = new Integer(this.get_w_r());
        toSend[3] = new Integer(this.get_preempted_val());
        if(value != null)
            toSend[4] = value;
        if(super.pushToQueue(toSend))
            this.modified = false;
    }
}

private input_Channel partner_ref = null;
private int w_s_copy = 0;
private int w_r_copy = 0;
private int preempted_copy = 0;

public void set_partner_smp(input_Channel partner) {
  this.partner_ref = partner;
  this.partner = new input_Channel();
}

protected synchronized void updateLocalPartner(output_Channel p) {
  p.w_s = this.w_s_copy;
  p.w_r = this.w_r_copy;
  p.preempted = this.preempted_copy;
  if (value != null)
    p.value = this.value;
}

protected synchronized void updateLocalCopy() {
  this.w_s_copy = get_w_s();
  this.w_r_copy = get_w_r();
  this.preempted_copy = get_preempted_val();
}

Referring to Listing A.15, the set_partner() method (line 9) defines an input_Channel object in the output_Channel class which is used to retain a copy of the communication statuses of the partner input channel. The methods described in line 10-32 are get and set methods of the output channel’s communication statuses and value, which is detailed in [61]. The refresh() method (line 33-40) resets the output channel’s communication statuses and value buffer to the initial condition. The gethook() method (line 41-51) procures the updated partner channel’s communication statuses and value. In the case of partner channel in the same subsystem, the partner channel’s communication statuses and value are accessed directly via the partner input_Channel object and copied to the input_Channel object in the output_Channel class, while in the case of partner channel in a different subsystem, the partner channel’s communication statuses and value are obtained by invoking the getBuffer() method), which accesses the partner channel’s communication statuses and value stored in the buffer of the GenericChannel class. The sethook() method (line 58-75) stores a copy of the output channel’s communication statuses and value (if any) to local variables in output_Channel class, which are accessible by the partner channel (in the case of partner channel in the same subsystem). In the case of partner channel in a different subsystem, the sethook() method makes an attempt to transmit the output channel’s communication statuses and value to the partner channel via link. The setDistributed() method (line 77) sets a flag in the output channel indicating that the output channel is used to communicate with a partner channel in a different subsystem. The set_partner_smp() method (line 81-84) creates a mechanism to allow the input channel to access the partner channel’s object and obtain the partner channel’s communication statuses and value (only applicable in the case of partner channel
in the same subsystem). The updateLocalPartner() method (line 93-99) obtains the updated partner channel’s communication statuses and value (if any) and stores them to the output_Channel object as a copy of the partner channel’s output_Channel object\textsuperscript{34}. The updateLocalCopy() method (line 100-104) updates the copy of the input channel’s communication statuses with the updated input channel’s communication statuses.

\section*{A.2.15 Scheduler}

The Scheduler class is a Java abstract class which acts as a template to implement scheduling policy functionality. The code snippet describing the Scheduler class is shown in Listing A.16. Referring to Listing A.16, the Scheduler class has three methods. The addClockDomain() method (line 5) includes a particular CD into the scheduling function for execution. The run() method (line 6) initiates the scheduling function to run. The tick() method (line 7-10) triggers the execution of individual CDs included in the scheduling function and also the channel routing function by invoking the run() method of the ClockDomain class and runInterfaceManager() method in the BaseInterface class, respectively.

Listing A.16. The Scheduler class code snippet.

\begin{lstlisting}[language=Java]
import systemj.bootstrap.ClockDomain;
import systemj.common.BaseInterface;

public abstract class Scheduler extends BaseInterface{
    public abstract void addClockDomain(ClockDomain cd);
    public abstract void run();
    public final void tick(ClockDomain cd){
        cd.run();
        super.runInterfaceManager();
    }
}
\end{lstlisting}

\section*{A.2.16 CyclicScheduler}

The CyclicScheduler class is a Java class which provides a CD scheduling function which is based on round-robin scheduling policy. The code snippet describing the CyclicScheduler class is shown in Listing A.17. Referring to Listing A.17, the CyclicScheduler class defines two methods. The addClockDomain() method (line 6-8) includes a particular CD to the scheduling function for execution. The run() method triggers the execution of individual CDs based on the round-robin scheduling policy (line 9-13).

\textsuperscript{34} Only used in the case of partner channel in the same subsystem with the input channel.
Listing A.17. The CyclicScheduler class code snippet.

```java
import systemj.bootstrap.ClockDomain;
import systemj.interfaces.Scheduler;

public class CyclicScheduler extends Scheduler{
    private Vector cdarray = new Vector();
    public void addClockDomain(ClockDomain cd) {
        cdarray.addElement(cd);
    }
    public void run() {
        for(int i=0;i<cdarray.size();i++){
            tick((ClockDomain)cdarray.elementAt(i));
        }
    }
}
```

A.2.17 GenericInterface

The GenericInterface class is a Java abstract class which acts as a template to implement links. The class inherits the BaseInterface class to have access to functionality provided in the BaseInterface class. The code snippet describing the GenericInterface class is shown in Listing A.18. Referring to Listing A.18, the GenericInterface class has four methods. The configure() method (line 4) is provided to implement the functionality to obtain link configuration for link instantiation. The invokeReceivingThread() method (line 5) is provided to implement Java thread which implement receiving function of link. The setup() method (line 6) is provided to implement function to copy the channel data passed by output channel to prepare for its transmission via link. The transmitData() method (line 7) is provided to implement the channel data transmission function of link.

Listing A.18. The GenericInterface class code snippet.

```java
import systemj.common.BaseInterface;
import systemj.common.InterfaceManager;
public abstract class GenericInterface extends BaseInterface{
    public abstract void configure(Hashtable ht);
    public abstract void invokeReceivingThread();
    public abstract void setup(Object[] o);
    public abstract boolean transmitData();
}
```

A.2.18 TCPIPInterface

The TCPIPInterface class is a Java class which implements link as TCP/IP communication. The class inherits the GenericInterface class as a link implementation. The code snippet describing the TCPIPInterface class is shown in Listing A.19.
Listing A.19. The TCPIPInterface class code snippet.

```java
import systemj.common.InterfaceManager;
import systemj.interfaces.GenericInterface;
import systemj.interfaces.GenericChannel;
public class TCPIPInterface extends GenericInterface implements Runnable {
    private String ip;
    private int port;
    private Object[] buffer;
    @Override
    public void configure(Hashtable ht) {
        if(ht.containsKey("Args")){
            String[] args = ((String)ht.get("Args")).trim().split(":");
            if(args.length != 2)
                throw new RuntimeException("Incorrect Args for TCP/IP interface : must be <IP>:<Port>");
            ip = args[0];
            port = new Integer(args[1]).intValue();
        }
    }
    @Override
    public void invokeReceivingThread() {
        new Thread(this).start();
    }
    @Override
    public void setup(Object[] o) {
        buffer = o;
    }
    @Override
    public boolean transmitData() {
        try {
            Socket client = new Socket(ip, port);
            ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
            out.writeObject(buffer);
            client.close();
        } catch(java.net.ConnectException e){
            System.out.println("Could not reach server "+ip+":"+port);
            return false;
        } catch(Exception e){
            e.printStackTrace();
            return false;
        }
        return true;
    }
    @Override
    public void run() {
        try {
            ServerSocket = new ServerSocket(port, 50, InetAddress.getByName(ip));
            while(true){
                Socket = serverSocket.accept();
                ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
                Object[] o = (Object[])ois.readObject();
                super.forwardChannelData(o);
                socket.setSoLinger(true, 0);
                socket.close();
            }
        }
    }
```
Referring to Listing A.19, the TCPIPInterface class has five methods. The configure() method (line 9-17) copies the link configuration to the class for its instantiation. The invokeReceivingThread() method (line 19-21) initiates the execution of the thread that performs the receiving function of the link. The transmitData() method (line 27-43) transmits channel data via link implemented as TCP/IP connection. The run() method (line 45-67) is invoked by the invokeReceivingThread() method to initiate the TCP/IP receiving function of the link.
Listing B.1. The code snippet describing the TransmitDisc class.

```java
public class TransmitDisc extends GenericSignalSender{
    DatagramSocket s1 = null;
    String CDName = null;
    @Override
    public void configure(Hashtable data) throws RuntimeException {
        if(data.containsKey("CDName")){
            CDName = (String) data.get("CDName");
        }
    }
    @Override
    public void run(){
        try{
            int recPort = getAvailablePort();
            DiscPortAssignment.SetDiscPortAssignment(CDName, recPort);
            while(!DiscPortAssignment.GetDiscReplyPortReady(CDName)){}
            String SSOrig = SJSSCDSignalChannelMap.getLocalSSName();
            if(!SJRegistryEntry.IsRegistryEntryEmpty()){  
                JSONObject jsReg = SJRegistryEntry.GetRegistryFromEntry();
                String regID="";
                String regAddr = "";
                Enumeration keysJSReg = jsReg.keys();
                while(keysJSReg.hasMoreElements()){
                    regID = keysJSReg.nextElement().toString();
                    regAddr = jsReg.getString(regID);
                    break;
                }
                TransmitDiscMsg(SSOrig, regID, regAddr, recPort);
            }
        } catch (JSONException jex){
            jex.printStackTrace();
        }
    }
    private int getAvailablePort() {
        int port = 3333;
        while(true){
            try {
                s1 = new DatagramSocket(port);
                break;
            } catch (IOException e) {
                port++;
            }
        }
        return port;
    }
}
```
private void TransmitDiscMsg(String SSOrigin, String regID, String regAddr, int recPort) {
    JSONObject js = new JSONObject();
    SJSOAMessage sjdisc = new SJSOAMessage();
    MulticastSocket s = null;
    try {
        s = new MulticastSocket(1990);
        String message = sjdisc.ConstructNoP2PServToRegDiscoveryMessage(SSOrigin, regID, recPort);
        InetAddress ipAddress = InetAddress.getByName(regAddr);
        byte[] msg = new byte[65507];
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(byteStream));
        out.writeObject(message);
        out.flush();
        msg = byteStream.toByteArray();
        out.close();
        DatagramPacket hi = new DatagramPacket(msg, msg.length, ipAddress, 1990);
        s.send(hi);
        s.close();
    } catch (Exception iex) {
        iex.printStackTrace();
        s.close();
    }
}

Referring to Listing B.1, the TransmitDisc class inherits the GenericSignalSender to describe the implementation of physical interface of output signal (line 1). The configure() method (line 5-9) provides necessary information for the initialization of the output signal’s physical interface, which in this case it obtains the CD name of the CD utilizing the output signal. The run() method starts by invoking the getAvailablePort() method (line 34-45) in line 13 to obtain any non-allocated physical UDP/IP port to receive Discovery Reply. The physical port information is then stored in a data structure named DiscPortAssignment which is accessible by the ReceiveDisc class for the ReceiveDisc class to implement the input signal’s physical interface to receive Discovery Reply (line 14). The run() method then attempts to synchronize with the ReceiveDisc class to instantiate the physical port used to receive Discovery Reply message via the DiscPortAssignment class (line 15-16). The run() method then accesses the data stored in the data structure of SJRegistryEntry to obtain the physical address and identifier of the global service registry for generating and transmitting Discovery message (line 19-27). Then in line 28, the run() method invokes the TransmitDiscMsg() method which also invokes the method to generate Discovery message and transmit the Discovery message (line 46-68). Note that the SOSJ RTS allocates UDP/IP port number 1990 for Discovery purpose. The physical port number obtained in line 13 is included in the created Discovery message for the global service registry to send Discovery reply through the corresponding physical port.
ReceiveDisc Class

Listing C.1. The ReceiveDisc class code snippet.

```java
public class ReceiveDisc extends GenericSignalReceiver{
    String CDName;
    DatagramSocket s1 = null;
    @Override
    public void configure(Hashtable data) throws RuntimeException {
        if(data.containsKey("CDName")){
            CDName = (String)data.get("CDName");
        }
    }
    @Override
    public void run(){
        while(!terminated){
            Object[] obj = new Object[2];
            if(DiscPortAssignment.IsDiscPortAssignmentExist(CDName)){
                JSONObject jsAllServs = new JSONObject();
                JSONObject jsMsg = new JSONObject();
                try {
                    int portNum = DiscPortAssignment.GetDiscPortAssignment(CDName);
                    jsMsg = ReceiveDiscReplyMsg(portNum);
                    if(!jsMsg.isEmpty()){
                        Enumeration keysServList = jsMsg.getJSONObject("serviceList").keys();
                        while(keysServList.hasMoreElements()){
                            String indivSSName = keysServList.nextElement().toString();
                            JSONObject jsAllCDStats = jsMsg.getJSONObject("CDStats");
                            JSONObject jsAllSSAddrs = jsMsg.getJSONObject("SSAddrs");
                            Enumeration keysAllCDStats = jsAllCDStats.keys();
                            InterfaceManager im = IMBuffer.getInterfaceManagerConfig();
                            while(keysAllCDStats.hasMoreElements()){
                                String indivSSNameCDStats = keysAllCDStats.nextElement().toString();
                                JSONObject jsCDStats = jsAllCDStats.getJSONObject(indivSSNameCDStats);
                                Enumeration keysJSCDStats = jsCDStats.keys();
                                while(keysJSCDStats.hasMoreElements()){
                                    String keyCDName = keysJSCDStats.nextElement().toString();
                                    im.addCDLocation(indivSSNameCDStats, keyCDName);
                                }
                            }
                            Enumeration keysAllSSAddrs = jsAllSSAddrs.keys();
                            while(keysAllSSAddrs.hasMoreElements()){
                                RegAllCDStats.AddCDStat(indivSSNameCDStats, jsAllSSAddrs.getJSONObject(indivSSNameCDStats));
                        }
                    }
                }
                Enumeration keysAllCDStats = jsAllCDStats.keys();
                while(keysAllCDStats.hasMoreElements()){
                    String indivSSNameCDStats = keysAllCDStats.nextElement().toString();
                    JSONObject jsCDStats = jsAllCDStats.getJSONObject(indivSSNameCDStats);
                    Enumeration keysJSCDStats = jsCDStats.keys();
                    while(keysJSCDStats.hasMoreElements()){
                        String keyCDName = keysJSCDStats.nextElement().toString();
                        im.addCDLocation(indivSSNameCDStats, keyCDName);
                    }
                }
            }
        }
    }
}
```

String indivSSNameSSAddr = keysAllSSAddrs.nextElement().toString();
RegAllSSAddr.AddSSAddr(indivSSNameSSAddr, jsAllSSAddrs.getString(indivSSNameSSAddr));
jsAllServs.put(indivSSName, jsServList.getJSONObject(indivSSName));

try {
    catch (JSONException ex) {
        ex.printStackTrace();
    }

    if (jsAllServs.length() > 0) {
        obj[0] = Boolean.TRUE;
        obj[1] = jsAllServs.toString();
    } else {
        obj[0] = Boolean.TRUE;
        obj[1] = "timeout";
    }
    super.setBuffer(obj);
}

public ReceiveDisc(){
    super();
}

private JSONObject ReceiveDiscReplyMsg(int recPort){
    JSONObject js = new JSONObject();
    try{
        s1 = new DatagramSocket(recPort);
        byte data[];
        byte packet[] = new byte[65507];
        DatagramPacket pack = new DatagramPacket(packet, packet.length);
        DiscPortAssignment.SetDiscReplyPortReady(CDName);
        s1.setSoTimeout(1000);
        s1.receive(pack);
        data = new byte[pack.getLength()];
        System.arraycopy(packet, 0, data, 0, pack.getLength());
        if(data.length > 0)
            {
                if(((int)data[0] == -84) && ((int)data[1] == -19))
                    {
                        try
                        {
                            ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(data));
                            Object mybuffer = ois.readObject();
                            js = new JSONObject(new JSONTokener(mybuffer.toString().trim()));
                        }
                    }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
                }
            s1.close();
        } catch (SocketTimeoutException stex){
            s1.close();
        } catch (IOException iex){
            iex.printStackTrace();
        }
}
Referring to Listing C.1, the ReceiveDisc class inherits the GenericSignalReceiver class to describe the implementation of physical interface of output signal (line 1). The configure() method provides necessary information for the initialization of the input signal’s physical interface, which in this case it obtains the CD name of the CD utilizing the input signal (line 5-9). The run() method starts with checking the data structure in the DiscPortAssignment to determine whether a UDP/IP physical port has been allocated to receive Discovery Reply message (line 14). If a physical port has been allocated, the run() method proceeds to invoke the ReceiveDiscReplyMsg() method (line 19). The ReceiveDiscReplyMsg() method is described in line 68-105. The ReceiveDiscReplyMsg() method starts by instantiating a UDP/IP physical socket with the port number fetched from the DiscPortAssignment class used to receive Discovery Reply message (line 71). Then, it invokes the SetDiscReplyPortReady() method of the DiscPortAssignment to notify the TransmitDisc class that the physical socket used to receive Discovery Reply message has been created and is ready to use (line 75). The method continues by waiting to receive Discovery Reply message from the global service registry (line 77) and this process will terminate after 1000 milliseconds to anticipate if the transmitted Discovery or inbound Discovery Reply messages are lost during transmission, allowing the Discovery message to be re-transmitted if necessary. Once a UDP/IP packet containing Discovery Reply message is received, the message is extracted from the packet and reconstructed (line 88). Then the information contained in the Discovery Reply is obtained. The information consists of service descriptions, CD macro-states, and physical location of subsystems. The extraction of service descriptions, CD macro states, and physical location of subsystems are performed in line 21, 25, and 26. Next, the CD macro state information is stored in the data structure of RegAllCDStats (line 32). The CD macro state information includes the association of CDs with their corresponding subsystems, thus it is also stored in the data structure of the InterfaceManager (line 34-38). Also, the physical location of subsystems information is stored in the data structure of RegAllSSAddr (line 44). Meanwhile, the service descriptions are returned (line 54) and become visible to the corresponding CD through the input signal SOSJDiscReply. In case no Discovery Reply is received, the implementation notifies the CD by returning the value ‘timeout’ (line 57). The InterfaceManager instance can be accessed and stored through a data structure class named IMBuffer. The code snippet describing the IMBuffer class can be found in [177].
User-Side Reconfiguration

To facilitate human users in triggering dynamic behaviours using SOSJ e.g. through a remote computing machine, a client application has been developed. The client application provides a graphical user interface (GUI) for the user to define the required parameters for reconfiguration. Any commands sent via the client application will be received by the Reconfig Req Receiver of the SOSJ RTS. Figure D.1 shows the GUI of the developed client application.

![Figure D.1. GUI of the SOSJ client application.](image)

The GUI allows the user to perform reconfiguration by sending commands which trigger dynamic behaviours from the drop down menu called Command. Available commands are CreateCD, SuspendCD, WakeUpCD, KillCD, and MigrateCD, which have identical purposes as the SOSJ Dynamic Function Calls described in Section 6.2. Each command requires different parameters which have to be provided by the user. The Transmit Command button needs to be pressed by the user to send command to initiate reconfiguration. Table D.1 describes each field in the SOSJ client application and their descriptions.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS Name</td>
<td>Subsystem identifier containing the CD to be reconfigured</td>
</tr>
<tr>
<td>SS Address</td>
<td>The physical location (e.g. IP address, URI, etc.) of the subsystem</td>
</tr>
<tr>
<td>CD Name</td>
<td>The CD identifier</td>
</tr>
<tr>
<td>CD Config</td>
<td>The CD configuration file</td>
</tr>
<tr>
<td>CD Service Description</td>
<td>The CD service description file</td>
</tr>
<tr>
<td>Migration Type</td>
<td>The type of the CD migration (strong or weak)</td>
</tr>
<tr>
<td>Migration SS Destination</td>
<td>The identifier of the destination subsystem of the migrating CD</td>
</tr>
</tbody>
</table>

Figure D.1 also shows an example of how the user can use the application and the fields to fill to create a CD named ‘CDBC1’, with CD configuration and service description described in the ‘cbcd1cdconfig.xml’ and ‘cbcd1cdsd.xml’ files (respectively), in the subsystem named ‘SS1’ with physical location of 192.168.1.4 (IPv4 address). Once the reconfiguration command is sent, it will be received by the **Reconfig Req Receiver** of the RTS of the subsystem the CD belongs to, and then the reconfiguration is performed following the CD creation procedure.
List of Publications

The work presented in this thesis is based on the following research articles.