INCREMENTAL CONSTRUCTION OF COMPONENT-BASED SYSTEMS: A STUDY BASED ON CURRENT COMPONENT MODELS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2015

By
Tauseef Rana
School of Computer Science
3 Incremental Construction in CBD

3.1 Incremental Construction

3.1.1 Incremental Construction Process

3.1.2 Component and System Behaviour

3.1.3 Behaviour Containment

3.1.4 Incremental Composition

3.2 System Construction Approaches

3.3 The Outline of the Study of Incremental Construction in CBD

3.3.1 Ways to Increment a System

3.3.2 Behaviour Containment

3.3.3 Fixed versus Not-fixed Behaviour of Components

3.3.4 Construction of an Expression Calculator

3.4 Object-based Component Models

3.4.1 Increment by Composition

3.4.2 Increment by Adding Code

3.4.3 Incremental Construction of The Expression Calculator

3.4.4 Discussion and Evaluation

3.5 Architecture Description Languages

3.5.1 Mainstream ADLs

3.5.2 Invasive ADL

3.5.3 Discussion and Evaluation

3.6 Component Models with Encapsulated Component

3.6.1 Web Services

3.6.2 The X-MAN Component Model

3.6.3 Discussion and Evaluation

3.7 A Comparative Study of the Three Categories of Component Models

3.7.1 Behaviour Based Comparison

3.7.2 A Component Model for Incremental Construction

3.8 A Review of Related Work

3.8.1 JavaBeans

3.8.2 ProCom Component Model

3.8.3 C2 Component Model

3.8.4 SOFA Component Model

3.8.5 Pluggable Software Units

3.8.6 Reo Connectors
4 The X-MAN Component Model

4.1 The Conventional Model

4.1.1 Atomic Components

4.1.2 Exogenous Connectors

4.1.3 Composition in X-MAN

4.1.4 System Construction

4.1.5 The Bank Example

4.2 The Extended X-MAN Component Model

4.2.1 Limitations of X-MAN

4.2.2 Extending X-MAN Components

4.2.3 Extending Exogenous Connectors

4.3 Presenting Exogenous Connectors

4.4 Unconstrained Exogenous Connectors

4.4.1 Invocation Connector

4.4.2 Infinite Loop Connector

4.4.3 Sequencer

4.5 Constrained Exogenous Connectors

4.5.1 Pipe

4.5.2 Guard Connector

4.5.3 Selector

4.5.4 Finite Loop Connector

4.6 The Bank Example

4.7 Exogenous Composition Framework

4.8 Chapter Summary

5 Incremental Construction Using X-MAN'

5.1 Towards Achieving Incremental Construction in X-MAN'

5.2 Adding Increments

5.2.1 By Composition

5.2.2 By Increasing the Connector Arity

5.2.3 By Adding Adaptors

5.2.4 Rule to Refine Adaptor Constraint

5.3 Refactoring

5.3.1 Replacing Sequencer by Pipe
5.3.2 Replacing a Connector by Many Connectors 176
5.4 The Construction Process 180
5.5 Extended Bank Example 182
5.5.1 System Use Case 182
5.5.2 System Construction 183
5.5.3 Summary and Discussion 190
5.6 Limitations of Our Approach 193
5.7 Chapter Summary 193

6 The Common Component Modelling Example 194
6.1 The System Overview 194
6.2 Extending The Construction Guidelines 196
6.2.1 Incrementing System by Component Sharing 197
6.2.2 Relocating an Adaptor 198
6.3 Extending The Exogenous Connectors 200
6.3.1 A Special Selector 200
6.3.2 Chain of Responsibility Connector 201
6.3.3 Extending The Finite Loop Connector 211
6.4 Cash Desk System 218
6.4.1 Use Case 1 (UC1) 219
6.4.2 Use Case 2 (UC2) 226
6.4.3 Summary 233
6.5 Chapter Summary 234

7 Conclusions and Future Work Recommendations 236
7.1 Conclusions 236
7.2 Future Work 238
7.2.1 Composite Connectors 239
7.2.2 Composition Connectors for Deployment 239
7.2.3 Dynamic Incremental Composition 239
7.2.4 Extending Flow Constraint Language 240
7.2.5 Extending The Exogenous Composition Framework Tool 240
7.2.6 Exception Handling 240
7.2.7 Supporting Concurrent Processing 241

Bibliography 242
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Chapter-wise contributions</td>
<td>23</td>
</tr>
<tr>
<td>2.1</td>
<td>Categories of component models</td>
<td>28</td>
</tr>
<tr>
<td>2.2</td>
<td>Components in current component models</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Containment</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Extension</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>Connection</td>
<td>32</td>
</tr>
<tr>
<td>2.6</td>
<td>Coordination</td>
<td>33</td>
</tr>
<tr>
<td>2.7</td>
<td>A taxonomy of software composition mechanisms</td>
<td>34</td>
</tr>
<tr>
<td>2.8</td>
<td>Composition of EJBs</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>Composite of Acme components</td>
<td>37</td>
</tr>
<tr>
<td>2.10</td>
<td>Invasive software composition</td>
<td>37</td>
</tr>
<tr>
<td>2.11</td>
<td>A bank system</td>
<td>39</td>
</tr>
<tr>
<td>2.12</td>
<td>Web service composition</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Incremental construction process</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Component and system behaviour</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>A specific example of Behaviour containment</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Behaviour containment</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>System construction approaches</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Components with fixed behaviour</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Class nesting, object composition and object aggregation</td>
<td>53</td>
</tr>
<tr>
<td>3.8</td>
<td>Class inheritance</td>
<td>54</td>
</tr>
<tr>
<td>3.9</td>
<td>Object delegation</td>
<td>54</td>
</tr>
<tr>
<td>3.10</td>
<td>Aspect weaving</td>
<td>55</td>
</tr>
<tr>
<td>3.11</td>
<td>Trait-class composition</td>
<td>55</td>
</tr>
<tr>
<td>3.12</td>
<td>Increment by adding repetition construct</td>
<td>56</td>
</tr>
<tr>
<td>3.13</td>
<td>A system to evaluate a mathematical expression</td>
<td>57</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>3.14</td>
<td>A filter component and a composite</td>
<td>60</td>
</tr>
<tr>
<td>3.15</td>
<td>A system to evaluate a mathematical expression</td>
<td>61</td>
</tr>
<tr>
<td>3.16</td>
<td>A component and a composite</td>
<td>62</td>
</tr>
<tr>
<td>3.17</td>
<td>A system to evaluate a mathematical expression</td>
<td>64</td>
</tr>
<tr>
<td>3.18</td>
<td>A component and a composite</td>
<td>65</td>
</tr>
<tr>
<td>3.19</td>
<td>A system to evaluate a mathematical expression</td>
<td>66</td>
</tr>
<tr>
<td>3.20</td>
<td>Components with provided ports only</td>
<td>67</td>
</tr>
<tr>
<td>3.21</td>
<td>Components with incompatible ports</td>
<td>67</td>
</tr>
<tr>
<td>3.22</td>
<td>A system to evaluate a mathematical expression</td>
<td>68</td>
</tr>
<tr>
<td>3.23</td>
<td>A system to evaluate a mathematical expression</td>
<td>70</td>
</tr>
<tr>
<td>3.24</td>
<td>A system to evaluate a mathematical expression</td>
<td>71</td>
</tr>
<tr>
<td>3.25</td>
<td>Behaviour based categories</td>
<td>73</td>
</tr>
<tr>
<td>3.26</td>
<td>Behaviour of Components</td>
<td>74</td>
</tr>
<tr>
<td>3.27</td>
<td>Composition of JavaBeans</td>
<td>78</td>
</tr>
<tr>
<td>3.28</td>
<td>System construction in C2</td>
<td>79</td>
</tr>
<tr>
<td>3.29</td>
<td>System construction in SOFA 2.0</td>
<td>80</td>
</tr>
<tr>
<td>3.30</td>
<td>A network pluggable unit</td>
<td>81</td>
</tr>
<tr>
<td>3.31</td>
<td>An example</td>
<td>82</td>
</tr>
<tr>
<td>4.1</td>
<td>Atomic component</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>Exogenous connectors</td>
<td>86</td>
</tr>
<tr>
<td>4.3</td>
<td>Composition connectors</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>Composition in X-MAN</td>
<td>90</td>
</tr>
<tr>
<td>4.5</td>
<td>The X-MAN system architecture</td>
<td>91</td>
</tr>
<tr>
<td>4.6</td>
<td>Deployment phase components</td>
<td>92</td>
</tr>
<tr>
<td>4.7</td>
<td>A simple bank system</td>
<td>92</td>
</tr>
<tr>
<td>4.8</td>
<td>A conceptual model of X-MAN components</td>
<td>95</td>
</tr>
<tr>
<td>4.9</td>
<td>Infinite loop in a system</td>
<td>96</td>
</tr>
<tr>
<td>4.10</td>
<td>Signature annotations for a component service</td>
<td>98</td>
</tr>
<tr>
<td>4.11</td>
<td>Service signature with constrained output parameters</td>
<td>99</td>
</tr>
<tr>
<td>4.12</td>
<td>A conceptual model of exogenous connectors</td>
<td>101</td>
</tr>
<tr>
<td>4.13</td>
<td>Service Propagation</td>
<td>102</td>
</tr>
<tr>
<td>4.14</td>
<td>Open composition connector</td>
<td>103</td>
</tr>
<tr>
<td>4.15</td>
<td>Open sequencer connector</td>
<td>103</td>
</tr>
<tr>
<td>4.16</td>
<td>Extended open composition connector</td>
<td>104</td>
</tr>
<tr>
<td>4.17</td>
<td>Connector and composite behaviour</td>
<td>105</td>
</tr>
</tbody>
</table>
4.18 Open composition connectors ............................. 106
4.19 Interface generation ...................................... 109
4.20 Control and Data Flow of IU .............................. 110
4.21 Interface generation ...................................... 111
4.22 Control and Data Flow of IL1 ............................. 113
4.23 Interface generation ...................................... 114
4.24 Examples ..................................................... 115
4.25 Control and Data Flow of SEQ1 ........................... 116
4.26 Common terms for EBNF ................................. 118
4.27 EBNF of FCL constraints for a Pipe ..................... 119
4.28 Example of a pipe constraint .............................. 120
4.29 Interface generation ...................................... 121
4.30 Interface generation by a pipe connector ................. 122
4.31 Control and Data Flow of PIPE1 ......................... 123
4.32 Group 1 of PIPE1 .......................................... 124
4.33 Transitions $T_4$ and $T_5$ ................................. 124
4.34 Transition $T_6$ .............................................. 125
4.35 Transition $T_7$ and $T'_7$ .................................. 125
4.36 Transition $T_8$ and $T_9$ .................................. 126
4.37 EBNF of FCL constraints for a guard ..................... 127
4.38 Example of a guard constraint ............................ 128
4.39 Interface generation ...................................... 129
4.40 Control and Data Flow of G1 ............................. 130
4.41 Group 1 of G1 .............................................. 131
4.42 Transitions $T_2$ and $T_3$ in Guard CP-net .............. 132
4.43 Transition $T_4$ in Guard CP-net ......................... 132
4.44 Transitions $T_4'$ and $T_4''$ in Guard CP-net .......... 133
4.45 Transitions $T_5$, $T_6$ and $T_6'$ in Guard CP-net ...... 133
4.46 Transitions $T_7$ and $T_8$ in Guard CP-net ............. 134
4.47 EBNF of FCL constraint for a selector .................... 135
4.48 Example of a selector constraint .......................... 136
4.49 Interface generation ...................................... 137
4.50 Control and Data Flow of SEL1 ........................... 138
4.51 Group1 of SEL1 ............................................ 139
4.52 Transitions $T_2$ and $T_3$ in Selector CP-net .......... 140
B.9 Simulation of control/data flows in a CoR connector . . . . . . . . . . 278
B.10 Simulation of control/data flows in a finite loop connector with a pa-
rameter mapping constraint . . . . . . . . . . . . . . . . . . . . . . 280
C.1 Extending service signatures . . . . . . . . . . . . . . . . . . . . . 281
C.2 The process of interface generation . . . . . . . . . . . . . . . . . . 282
C.3 Extending service signatures . . . . . . . . . . . . . . . . . . . . . . 283
C.4 Interface generation . . . . . . . . . . . . . . . . . . . . . . . . . . 284
D.1 $S_0$, $S_1$, $S_2$, $S_3$, and $S_4$ . . . . . . . . . . . . . . . . . . . . . . 285
D.2 $S_5$, $S_6$, $S_7$, and $S_8$ . . . . . . . . . . . . . . . . . . . . . . . . . 286
D.3 $S_9$ and $S_{10}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 287
D.4 $S_{11}$ and $S_{12}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 287
D.5 $S_{13}$, $S_{14}$, and $S_{15}$ . . . . . . . . . . . . . . . . . . . . . . . . . . 288
D.6 $S_{16}$ and $S_{17}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 289
D.7 $S_{18}$ and $S_{19}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 290
D.8 $S_{20}$, and $S_{21}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 291
D.9 $S_{22}$, and $S_{23}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292
D.10 $S_{24}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 293
Abstract

Incremental system construction aims to build systems iteratively by adding increments in a stepwise manner, starting from a small initial system. Such an approach holds the promise of managing scale and complexity, and should therefore be useful for building large systems. In the context of software development, many approaches based on refinement have been proposed for incremental system construction. In general, these approaches are top-down and not bottom-up.

The view taken in this thesis is that, incremental construction is intrinsically bottom-up, and would therefore be easier to achieve by means of Component-based Development (CBD), which is also intrinsically bottom-up. In this thesis, we investigate how incremental system construction can be achieved in CBD. We study incremental construction in the context of the three main categories of current component models, namely models where components are: (i) objects, (ii) architectural units, and (iii) encapsulated components.

By studying and comparing the strengths and weaknesses of these three categories of component models with respect to our notion of incremental construction, we show that incremental construction can be achieved most easily with component models with encapsulated components. We propose the construction guidelines to achieve incremental construction by using our extended X-MAN component model. In order to demonstrate the feasibility of our approach, we apply it to construct the cash desk system in the CoCoME example.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgments

I start by thanking The Creator (Allah) for granting me the resources and wisdom to be where I am today. Next, my words of gratitude are due to the National University of Sciences and Technology (NUST), Pakistan for sponsoring my PhD studies. I am also grateful to The University of Manchester for awarding me an Overseas Fee Bursary.

The last Prophet Muhammad (peace be upon him) said, “Whosoever is not thankful to the people, then he is not thankful to Allah.” I would like to thank my PhD supervisor Dr. Suzanne Embury for providing me with the opportunity to learn how to conduct meaningful research. My special thanks go to my PhD advisor Dr. Jonathan Shapiro for his righteous advices and to Dr. John Gurd for reading my thesis. I would also like to thank Dr. Richard Banach for his useful suggestions to improve my thesis. Next, without naming, I would like to thank everyone in my research group for their helpful discussions and suggestions for improving my work. I also would like to acknowledge the patience of my family for understanding my financial and time constraints.
Publications


Chapter 1

Introduction

In order to cope with scale and ever-increasing complexity, constructing software systems iteratively and incrementally is a promising approach. For software construction, the idea of iterative and incremental development is not new [77, 59]. Generally, incremental software construction is known as an evolutionary process for building a system iteratively; an iteration includes adding new functionality, and modifying and/or deleting an existing functionality of the system [13]. For software construction, in practice, there are a number of incremental development methods/processes to deal with systems with incomplete, imprecise and evolving requirements. For example, in agile and evolutionary software development methods [94, 91], the existing functionalities of a system may be changed or deleted according to changes in requirements. As a process, an incremental and iterative construction approach has a number of natural benefits, e.g. incorporating a change in the system is easier when compared to linear and one-shot (big-bang) construction approaches.

1.1 Incremental Construction

In this thesis, we¹ focus only on growing the system in increments, i.e. incremental construction, rather than addressing the whole process of “feedback-driven refinement with customer involvement and clearly delineated iterations” [59]. In this thesis, an iterative software construction approach is considered incremental if the functionality of the system under construction is incremented in each successive iteration. This means that in the construction process functionalities are only added, and not altered or deleted. Such an approach is applicable to construct systems with fixed requirements.

¹Throughout the thesis, the first person plural (we, our, us, etc.) is used to refer to the author alone.
or specifications. Primarily, software development/construction approaches are either a top-down approach or a bottom-up approach.

In top-down construction approaches [34, 104, 77], a top level abstract program with intended functional interface (and no code) is created as an initial step; for example, a program to calculate pay for an hourly paid worker begins as an empty procedure with a signature. The top level program is then decomposed and implemented iteratively. In each iterative step, a program is divided into more specific sub-level programs with full or partial functional implementation (by coding or by reuse). This way of decomposing the abstract sub-level programs is continued until the program is fully functional. This way of system construction is a flexible approach, because of the provision of adding new code anywhere in the program.

In software engineering, many approaches to incremental construction have been proposed, all based on the notion of stepwise refinement [34, 104], i.e. building a complex program from a simple program by adding features incrementally. In general, stepwise refinement means adding new functionality to a program in steps, as well as modifying existing functionality to achieve program correctness as per the program specifications [14].

Some incremental construction approaches focus on specification refinement [13, 1]. In these approaches, a specification is refined by another specification, such that the later satisfies the former. In the end, the code for the whole system is generated [1]. A specification component can also be refined/replaced by an implementation component in [13]; a system with only implementation components is executable.

Some approaches based on stepwise refinement focus on code refinement. In these approaches, an initial skeleton program is constructed, and then one feature is added to it at a time, until the program contains all the desired features. For example, in the software product line [18, 17], a software-system generator generates a large family of system programs by composing feature programs in many different ways. For a software product family, all features (base and increment) needed for a product line are developed by following a top-down approach. Constructing a system by feature composition is only allowed in a domain restricted way.

In stepwise refinement approaches, an increment and its composition with the system is not fixed. In contrast, in feature-oriented construction, although the increments are fixed, the composition is domain restricted. However, none of these approaches is truly component-based. As these approaches are top-down, a system construction begins either with an abstract program specification or a top-level (skeleton) program. In
1.2. WHAT IS A SOFTWARE COMPONENT MODEL?

contrast, in truly component-based development (CBD) approaches, system construction is a bottom-up process, i.e. starting system construction from pre-built system-independent components [52, 29].

1.2 What is a Software Component Model?

The idea of a component was first introduced by McIlroy in [74], where a single routine was considered as a component. The trend of software reuse has increased the importance of software composition. In the most general terms, software composition [81] refers to the composition of software components into larger composite components. The primary motivation for software composition is the reuse [89] of existing components.

The emphasis of many development paradigms, e.g. the object-oriented paradigm, is more on programming than on composition [80]. In contrast, for more emphasis on composition, component-based software development (CBD) represents a paradigm shift in software development. In CBD, there are many different software component models for component-based system construction. A software component model defines: (i) a component (functional program unit) as a unit of composition and (ii) composition (mechanism) to compose components together. For component models in CBD, it is desirable to have software units and composition mechanisms that support automated and systematic system construction [68]. In general terms, for CBD, composition can be defined as any possible and meaningful interaction between the software components involved.

1.3 Thesis Contributions

In this thesis, in order to construct component-based systems incrementally and iteratively, we redefine the notion of incremental construction with an emphasis on the behaviour containment. This redefinition of incremental construction enables us to reason about the preservation of existing functionality of a system after adding an increment to the system. Next, in the context of incremental construction, we study current component models to investigate how incremental construction can be achieved in the three main categories of current component models, namely models where components are: (i) objects, (ii) architectural units, and (iii) encapsulated components. This study has not been presented in the literature before. By studying and comparing the
strengths and weaknesses of component models from the three categories, we select the X-MAN component model [67] for its comparative suitability for incremental construction.

Having a component model that can be used for incremental construction is only half the story. The other half concerns the construction guidelines which include the ways to increment the system under construction. Such guidelines should also include suggestions to refactor the system if adding increments is not possible.

In order to further support incremental construction, we extend the conventional X-MAN component model and propose the incremental construction guidelines for our extended model (referred to as X-MAN'). By ‘conventional X-MAN’ we mean the version of X-MAN as presented in [67, 71]. To add behaviour to a system, for our proposed guidelines, we identify the set of all possible increments with respect to the proposed extensions to the X-MAN component model\(^2\). During system construction, to add new increments to an under construction system, the system may be required to be refactored. Hence, for the incremental construction guidelines, we propose a set of recommended changes in the system architecture by refactoring. In concept, the construction guidelines in our approach correspond to gradual evolution and larger refactoring (or simply refactoring) in [12]. To complete the construction guidelines for incremental construction, we extend the iterative construction process from [62] to refactor a system before adding an increment. In order to demonstrate the feasibility of our approach, we construct the cash desk system of the CoCoME example [88]. This implementation is evidence that our approach can be used to build practical and complex systems. Moreover, we learn from this exercise that existing exogenous connectors in X-MAN component model are not enough to build all kind of systems. Hence, we propose two new exogenous connectors.

Incremental construction only allows the addition of new functionalities to the existing composite (system) without losing any existing functionalities. Constructing a system incrementally, from fixed requirements or specifications, also has a clear advantage of incremental system testing. Using X-MAN', during system construction, intermediate composites with fixed behaviour can be tested. For this thesis, we developed a prototype tool (exogenous composition framework (ECF)) to support system construction. The semantics of our X-MAN' component model is implemented in ECF.

\(^2\)This set was identified as part of work carried out jointly with my colleagues Lau, Ng and Tran [62].
1.4. THESIS STRUCTURE

The overall contribution of this thesis is to define incremental construction and to study current component models for achieving incremental construction. However, the main contribution is to achieve incremental construction by using our X-MAN’ component model. In Figure 1.1, we show the chapter-wise contributions of this thesis.

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>1) Taxonomy of software composition mechanisms in CBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 3</td>
<td>2) Defining incremental construction and presenting a study based on the three categories of component models</td>
</tr>
</tbody>
</table>
| Chapter 4 | 3) Proposing to extend X-MAN for incremental construction  
4) Defining behaviour of exogenous connectors  
5) Implementation of X-MAN’ semantics in ECF |
| Chapter 5 | 6) Proposing the construction guidelines for X-MAN’ that include:  
a) ways to add increments to a system  
b) ways to refactor the system architecture to support incremental construction  
c) the construction process |
| Chapter 6 | 7) Proposing further extensions to X-MAN’  
8) Extending the construction guidelines proposed in Chapter 5  
9) Demonstrating the feasibility of our approach by constructing the cash desk system of the CoCoME example in X-MAN’ |

Figure 1.1: Chapter-wise contributions

1.4 Thesis Structure

In Chapter 2, we introduce the component-based development (CBD) paradigm. As composition is the essence of CBD, four general categories of composition mechanisms (containment, extension, connection and coordination) are presented\(^3\). Next, we propose the taxonomy of software composition mechanisms for the three categories of component models. From this taxonomy, we conclude that composition mechanisms from the containment and the extension categories are useful in creating a program unit. In contrast, composition mechanisms from the connection and the coordination categories are useful for creating systems; these two are the dominant composition mechanisms in CBD. This conclusion has not been presented in the literature before. Lastly, to illustrate the connection and the coordination composition mechanisms from the taxonomy, we describe composite examples in representative models from the three categories of component models. The knowledge gained from these example models will be used for the study of incremental construction in CBD.

\(^3\)A joint effort with another researcher in [68].
In Chapter 3, for CBD, we redefine the notion of incremental construction (with an emphasis on the behaviour containment). With this new definition of incremental construction, we are able to ensure behaviour preservation of existing system into the incremented system. To achieve incremental construction, we study the current component models from the standpoint of the three categories of components. In architectural description languages, based on different types of components and different types of port connection, we identify three (obvious) general architectural styles: (i) pipe-and-filter, (ii) client-server and (ii) publish-subscribe. In our study, we identify possible increments and mechanisms to add these increments to a system in the component models of each category (and sub-category). Based on our study, with respect to fixed behaviour of components, we categorise the studied component models into two groups. We conclude the chapter by selecting the X-MAN component model as the most suitable model to achieve incremental construction.

We present the X-MAN component model in Chapter 4. This is where we give all the necessary details of the existing model. In this chapter, for system construction, we identify a number of issues/limitations of the X-MAN component model. In order to construct practical systems incrementally, we extend the conventional X-MAN component model to address the identified issues/limitations of the model. The extended X-MAN is referred to as X-MAN'. For X-MAN' components, our proposed extensions include distinguishing components for user interactions from the components with computational behaviour, introducing annotations for services in a component’s interface, and the constrained output parameters for a service. For exogenous connectors in X-MAN', our proposed extensions include the concept of service propagation and behaviour containment, open composition connectors and the flow constraint language. In the conventional X-MAN, the semantics of exogenous connectors defined in [66, 82, 100] only address the control flow of composition connectors. In this chapter, we define the static and dynamic semantics of all exogenous connectors with our proposed extensions. In order to define the static semantics of exogenous connectors with the proposed extensions rigorously, we define interface generation (in the deployment phase) by exogenous connectors schematically. Moreover, for dynamic semantics (control and data flows in the run-time phase) of exogenous connectors, we use Coloured Petri net (CP-net) [53]. Lastly, we briefly introduce our tool (exogenous composition framework (ECF)) that we constructed for this thesis. With the help of this tool for the X-MAN' semantics, it was possible to demonstrate that our incremental construction approach with our X-MAN' model is achievable.
In Chapter 5, the construction guidelines for X-MAN’ are proposed which include possible ways to add increments, suggestions for refactoring the system under construction and the construction process. The primary tactic of the construction process is to identify a behavioural element from requirements and to add the identified element to the system. Behaviour identification and adding increments to the system continues iteratively until the system is complete. Throughout this chapter, to show our proposed construction guidelines in practice, we use a simple banking example. Further to illustrate the iterative construction process, we use the extended bank example. Finally, the extended bank example is constructed and simulated in ECF.

In Chapter 6, in order to demonstrate the feasibility of our approach, we construct the cash desk system of CoCoME (the common component modelling example) example from [88] by using X-MAN’. In order to construct the cash desk system by using X-MAN’, we propose further extensions to exogenous connectors. Moreover, construction guidelines for X-MAN’ are also extended. In the end of this chapter, from the constructed system, we show the contribution of different mechanisms of adding increments from our construction guidelines.

Finally, we conclude and set directions for future research work in Chapter 7.
Chapter 2

Component-based Development

In this chapter, the background material for the study of incremental construction in current component models is presented. In CBD, with respect to the interaction points of components, software component models are categorised into three groups, namely models where components are: (i) objects, (ii) architectural units, and (iii) encapsulated components. As composition is the essence of component models, a survey of composition mechanisms is presented for composing the aforementioned three kinds of components. Lastly, to illustrate composition mechanisms from the connection and coordination categories, simple composites are briefly described in well known representative component models from the three categories of component models.

2.1 Reuse-Oriented Approach

The concept of a software component is not new; the trend of software reuse has increased the importance of reusable software components and the mechanisms to compose components. Software reuse is a process of creating new software from existing software rather than developing from scratch [58]. Software reuse, a simple but effective technique for reducing the cost of software development, appears in many forms from ad-hoc (white-box) to systematic (black-box) software development approaches [89].

Software reuse appears at many different levels of solution development, such as at code level by reusing programming language constructs (selection, sequence and looping) [34], at function/service level, and at data structure level. The next level of reuse is the application of domain specific components [78]; for example, objects or frameworks for library management system. Through its implementation in a programming
language, a piece of software (component) is a program unit encompassing effort of
analysis, design, coding and testing as its intellectual property; therefore reusing a
tested component means more than simple code reuse.

For supporting development for/with reuse [70], component-based development
(CBD) [51, 97] represents a paradigm shift in software development. In suggestions
for research directions in software composition [80, 81], the importance of component
composition is highlighted. A component is a unit of reuse with defined interfaces
(plugs) to connect with other components to form a composite, thereby offering in-
creased reusability by composition as compared to other (non-component based) soft-
ware development paradigms. For CBD, software reuse is of course a fundamental ob-
jective to reduce the software production cost. However, in addition, CBD also seeks
to automate composition as much as possible [68], so as to reduce time-to-market as
well.

Nevertheless, component models in CBD add another dimension to the conven-
tional concept of reuse (of computational behaviour) and that is to reuse the explicit
composition operators (e.g. exogenous connectors in X-MAN). In CBD, a composi-
tion operator defines control/data flow policies as separate units from the components
(functional software units) [75, 26, 68]. In CBD, software components provide large
scale reuse of its intellectual property and in turn offers reduced development and
maintenance costs.

In the context of CBD, components as well as connectors can be reused indepen-
dently. Hence, a system construction from the pre-built elements (component and
connectors) [73, 29] is referred to as construction by composition. The result of this
composition is a composite or a system.

2.2 A Survey of Software Component Models

Lau and Wang in [72] suggest a reference framework which is universally applicable
to software component models. According to their framework, a component model
should define: (i) the syntax of components, (ii) the semantics of components and (iii)
the composition of components. The syntax of a component defines how components
are constructed and represented. The semantics of a component define what compo-
nents mean and their functional roles in a system. The composition of components
define how components are composed or assembled.
In the survey of component models [72], a number of component models (Enterprise JavaBeans (EJB) [33], JavaBeans [76], Component Object Model (COM) [35], .Net [103], CORBA Component Model (CCM) [50], Web Services [6], Fractal [23], Acme-like architecture description languages (ADLs) [28], UML2.0 [83], Kobra [10], Koala [101], SOFA [86], and PECOS [79, 48]) are analysed with respect to the syntax, semantics and composition of components. In this survey, based on component types, the studied component models are categorised into two categories, as shown in Figure 2.1.

<table>
<thead>
<tr>
<th>Components</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>JavaBeans,EJB,COM,.Net,CCM, Web Services, Fractal</td>
</tr>
<tr>
<td>Architectural Units</td>
<td>Acme-like ADLs,UML2.0,Kobra, Koala,SOFA,PECOS</td>
</tr>
</tbody>
</table>

Figure 2.1: Categories of component models

Next, based on the composition of components in the design and deployment phases, a taxonomy of component models is proposed (not shown here). In order to achieve reuse, composition is the essence of component models which is used to create composite components and systems. In a composite, components use message passing mechanism to interact with each other; message passing induces close coupling between communicating components which hampers component reuse.

### 2.3 Components in Current Component Models

In order to overcome the close coupling problem in component reuse described in Section 2.2, the X-MAN component model is defined in [71]. In this model, components encapsulate computation and data [69], and exogenous connectors encapsulate the control (and data) flow for the connected (composed) component(s). In X-MAN, a component has a provided interface and exogenous composition connectors define composition mechanisms for components.

In CBD, a generic component (Figure 2.2(a)) is a program unit with interfaces for provided and required service(s). These interfaces represent points of interactions for a component. In [68], based on public/provided interfaces (or services) and required interfaces, current component models are categorised into three main groups, namely
models where components are: (i) objects, (ii) architectural units and (iii) encapsulated components. Provided/required services can also be referred to as provided/required ports.

![Diagram of components in current component models]

Figure 2.2: Components in current component models

An object (Figure 2.2(b)) developed in an object-oriented language has public methods and external method calls (in the computation code of public methods). Public methods correspond to the provided services of the generic component and external method calls correspond to the required services of the generic component.

Components in architectural description languages are referred to as architectural units (Figure 2.2(c)). An architectural unit has in-ports and out-ports; ports are interaction points (or channels) for communication (by passing data/control) with the architectural unit.

The encapsulated component category (Figure 2.2(d)) includes X-MAN components and web services. Web services and components in the X-MAN component model do not have required interface. The provided interface of X-MAN components and WSDL (web service description language [27]) of web services corresponds to the provided service of the generic component. WSDL of a web service is available from a UDDI (universal description, discovery and integration) registry. Hence, we include web services in the category of encapsulated components rather than in the category of objects (as proposed in [72]).

2.4 A Survey of Software Composition Mechanisms

In general, software composition [81, 97] refers to the composition of software constructs into larger composite constructs. The primary motivation for software composition is reuse [89]. In current component models, a composition mechanism defines how two or more components are composed to yield a composite; the structure and the behaviour of the composite is defined by the composition mechanism in the component model.
CHAPTER 2. COMPONENT-BASED DEVELOPMENT

In different development paradigms, there are clearly many different composition mechanisms [22, 80, 93, 89, 56, 97, 96, 6, 36, 9] to compose different kinds of software constructs. Here, we consider a survey of software composition mechanisms in software development [68]. In this survey of software composition mechanisms, as data can be part of behavioural program units, only units of composition that define behaviour are considered, rather than constructs that define primitive types or pure data structures. Composition mechanisms compose units of composition into larger pieces of software, i.e. they compose pieces of behaviour into larger pieces of behaviour. In our view, a composition mechanism is at least binary in arity.

Components in many component models (e.g. Enterprise JavaBean (EJB), JavaBean, COM etc. [72]) are objects developed in an object-oriented programming language. For example, using Java, an EJB component can be created by using object aggregation/composition or inheritance mechanisms for objects. COM component model supports containment composition mechanism [95]. In the object-oriented development paradigm, there are a number of composition mechanisms for creating a bigger composite class by composing two or more classes and for incrementing a class with the behaviours of aspects [56], mixins [22], and traits [36]. In object-oriented development paradigm, program units are created (by coding) and further composed to create bigger program units. Hence, to make our study of achieving incremental construction independent of any specific object-based component model, the object-oriented development paradigm itself is considered as a component model. Such a consideration is in compliance with the reference framework for component models defined in [72].

In the context of three categories of components, from the survey of composition mechanisms, we focus on composition mechanisms from current component models as well as composition mechanisms to compose aforementioned constructs with objects. Hence, in this section, general categories of composition mechanisms from [68] are presented.

2.4.1 General Categories of Composition Mechanisms

Here, four general categories of composition mechanisms (containment, extension, connection and coordination) are presented. Unified modelling language (UML [83]) is widely known and used modelling language in software development; UML supports composition mechanisms for objects and architectural units. Hence, these general categories of composition mechanisms are compared with the corresponding UML notations. Existing composition mechanisms, composing two components or a program
2.4. A SURVEY OF SOFTWARE COMPOSITION MECHANISMS

unit with a component, fall into one of these four general categories [68].

**Containment**

Containment composition mechanism refers to defining a program unit by putting at least two other program units inside the definition. As illustrated in Figure 2.3(a), unit U3 is defined by putting two program units U1 and U2 in the definition of U3. The behaviour of the composite or the container unit U3 is defined in terms of the behaviour of the contained units U1 and U2. The precise nature of the containment differs from mechanism to mechanism.

![Figure 2.3: Containment](image)

Containment composition mechanism in standard UML notation is defined for classes only; the only forms of containment are object composition and object aggregation. As illustrated in Figure 2.3(b), a class U3 is defined by the composition/aggregation of two classes U1 and U2 in the definition of class U3. In UML, there is no notation for nested class definition. Compared to UML, the notion of containment covers nested definitions of classes, as well as object composition and object aggregation.

**Extension**

Defining the behaviour of a program unit by extending the behaviour of at least two other program units is referred to as an extension composition mechanism. In extension, the behaviours of the composed program units become the behaviour of the composite unit. As illustrated in Figure 2.4(a), unit U3 is defined by the extension of units U1 and U2. The behaviour of U3 is defined in terms of the behaviours of U1 and U2.

In UML, multiple inheritance is the only extension composition mechanism defined to compose classes. As illustrated in Figure 2.4(b), class U3 is defined by inheritance
from classes U1 and U2. The behaviour of U3 is defined in terms of the behaviours of U1 and U2. Compared to UML, the general notion of extension from [68] covers more composition mechanisms. Examples of composition by this notion includes multiple inheritance in object-oriented programming, aspect weaving [55] in aspect-oriented programming [56] and mixin-class inheritance in [22].

**Connection**

Defining a program unit by interaction (message passing) between the behaviours of two or more program units is referred to as the connection composition mechanism. As illustrated in Figure 2.5, there are two kinds of connection: (i) direct message passing (Figure 2.5(a)) and (ii) indirect message passing (Figure 2.5(b)). Plugs (correspond to ports) represent the channel for message communications between the program unit and its environment.

The connection mechanism induces tight coupling between the connected program units that send messages to each other by behaviour invocations. For interaction between the composed program units, direct message passing connection is referred to as (object) delegation, while indirect message passing connection is referred to as a connector.
Compared to UML, the notion of connection covers more composition mechanisms. In UML, connection is only defined for components and not for classes. UML has no notation for object delegation. In UML, the association notion between two classes can express a relationship in general, but not specific to method calls/invocations between two objects.

**Coordination**

Defining a program unit by coordinating the behaviours of two or more program units is referred to as coordination composition mechanism. As illustrated in Figure 2.6, program units U1 and U2 are coordinated by a coordinator. The coordinator communicates with the composed program units via a control and/or a data channel. There is no direct communication between the composed units. In contrast to connection, the coordination composition mechanism does not induce coupling between the composed program units. Coordination models and languages are classified as either data-driven or control-driven in [85]. Data coordination and control coordination can be considered as two kinds of coordination.

![Figure 2.6: Coordination](image)

In UML there is no notion of coordination, and hence no notation for coordination.

### 2.4.2 A Taxonomy of Software Composition Mechanisms

In the context of CBD, focusing on the composition mechanisms for the three categories of components (and for other program units which can be composed with these components), we propose a taxonomy of software composition mechanisms (an extracted survey of software composition mechanisms from [68]). As components in
object-based component models are developed in object-oriented programming languages, in this taxonomy, we consider composition mechanisms for program units (e.g. mixin, trait, aspect etc.) that can be composed with an object. Using the four general categories of composition mechanisms (from Section 2.4.1), our proposed taxonomy of software composition mechanisms in CBD is shown in Figure 2.7.

In the light of this taxonomy, we can conveniently conclude that the composition mechanisms from containment and extension categories, by and large, are used to define larger basic program units from the smaller program units. For instance, class inheritance builds the structure of an object (a unit of execution). Such a composition mechanism does not have any direct architectural function (in object communication) in a system [46].

In contrast, composition mechanisms from connection (except trait-class composition) and coordination categories define interactions amongst the composed program units in the resultant composite unit. For example, objects are connected by object delegation. Architectural units and fragment boxes are connected by port connection and invasive composition respectively. Control coordination defines interaction in a composite of web services and in a composite of X-MAN components. Hence, for defining interaction amongst the composed components, connection and coordination are the pre-dominant composition mechanisms in CDB [72]. Composition mechanisms that define interactions between the composed program units are useful to construct systems [12].

Hence, in the light of aforementioned conclusion from the survey of composition
mechanisms in CBD, our taxonomy of composition mechanisms is based on two categories: (i) mechanisms to create basic units and (ii) mechanisms to create systems.

### 2.5 Composites in Component Models

In CBD, a generic development process is comprised of component development process and system development process [52, 29, 70]. In the system development process, pre-developed components are composed or assembled to create a composite/system. In this section, by using well known representative component models, we consider simple composite/system examples to illustrate composition mechanisms from the connection and coordination categories (from Figure 2.7).

#### 2.5.1 Object-based Component Models

In this section, to illustrate object delegation from Figure 2.7, we select Enterprise JavaBeans (EJB) [33] as a representative component model. There are other object-based component models too, e.g. COM, CCM, and Fractal.

EJB is a component model in which components are beans (Java classes) hosted by an EJB container in a J2EE compatible server. EJB components are server-side program units which encapsulate the business logic for applications developed in Java. EJB components execute in the EJB container on an EJB compatible server (e.g. J2EE server).

In Figure 2.8, we show a simple composite of four enterprise JavaBeans connected by object delegation. Composition of these components is defined during programming. For example, method ‘a’ of component A has method call statement for method ‘b’ of component B. This kind of composition gets materialised once components are added to the EJB container in the right order. In a composite, the provided ports (public methods) of all components automatically become the provided ports of the composite.

#### 2.5.2 Architecture Description Languages

In CBD, based on components, the majority of component models are placed in the category of architectural units [72]. In general, a component model, in which components are architectural units, is referred to as an architectural description language (ADL) [16, 28]. An architectural unit is a computational program unit with in and out
communication channels (ports); a type signature is associated with each port. Architectural units are composed by connecting ports with matching type signatures.

In this section, to illustrate port connection and invasive composition from Figure 2.7, we consider two component models, Acme [45] and a component model with invasive software composition (ISC) [9].

**Acme**

In Acme, a component represents a program unit (with computational behaviour) or a data store. Access to the data and computational operations are only allowed through the specified ports of the component. A component can have multiple ports; a port is a point of interaction between the component and its environment.

Architectural units are composed by port connections between the matching ports. A connection is referred to as a connector which represents message passing (procedure call), event broadcasting, database queries and pipes [46]. In Acme, connectors also have interfaces with named roles to match with the ports of the components for connection. A binary connector (shown in Figure 2.9) have two roles, e.g. caller and callee in RPC (remote procedure call) connector. Some connectors may have more than two roles, e.g. an event broadcast connector has one event-announcer and arbitrary number of event-receiver roles.

In Acme, a system represents a configuration of components and connectors. In Figure 2.9, a simple system of two Acme components is shown. The client component has only one provided port and the server component has only one required port. Two components are connected by a binary RPC connector with caller and callee roles. In the system, the client initiates a request that is passed to the server by the connector and similarly the connector returns the response of the server to the client.
A Component Model with Invasive Composition

The model of invasive software composition [9] is based on three key elements: (i) fragment boxes, (ii) hooks and (iii) composer programs. A fragment box (Figure 2.10) has computational services and a composition interface (set of implicit and declared hooks). Figure 2.10 shows a composition of two components client and server by invasive composition. ISC supports fragment boxes with in/out data ports and with provided/required methods; such fragment boxes with declared data/method ports correspond to the architectural units in ADLs.

Implicit hooks (e.g. entry point of a method) are part of every fragment box; these implicit hooks represent the default composition interface of a component. In contrast, a declared hook pointing to a location in the code (e.g. a position in a loop construct in
a component’s implementation) is specified by the component designer/developer. For composition, composer programs transform the hooks to change the fragment box invasively. A declared hook disappears from the component if transformed. Conversely, implicit hooks can be transformed more than once (by different composers) and still be available for further transformations in the composite.

2.5.3 Component Models with Encapsulated Component

In this category, components have provided interface only. To illustrate orchestration and exogenous composition from Figure 2.7, we consider two component models the X-MAN component model [71] and web services [6]. Here, we describe the X-MAN component model briefly, for further details see Chapter 4.

The X-MAN Component Model

In the X-MAN component model, a component has only one provided interface and no required interface. Two or more components are composed by exogenous composition connectors. The composition mechanism to compose components is control coordination, as described in Section 2.4.1. Moreover, X-MAN defines adaptor connectors (e.g. guard and loop) for system construction. In X-MAN, for system construction, components as well as primitive exogenous connectors are pre-built elements stored in the respective repositories. In order to illustrate the roles of different exogenous connectors, a simple bank system is shown in Figure 2.11.

The bank system (in Figure 2.11) has one ATM (Automated Teller Machine) to serve two branches of a specific bank. During the system execution, the ATM component reads an ATM card and gets the authentication from the central bank. After authentication, PIPE1 (a composition connector) transfers control (and data) to G1 (a guard connector). The guard connector lets the control (and data) flow pass thorough if the card authentication was successful. Based on the bank details, a selector connector (SEL1) passes the request to one of the bank branches. After serving one customer, loop connector L1 repeats the execution to serve the next customer.

Web Services

In service-oriented computing, web services are basic elements of a distributed application [6]. A web service (running on a web server) may exhibit one or more web procedures with computational behaviour. A web service is made public by uploading
its interface specified in web service description language (WSDL) [27] on a UDDI (universal description, discovery and integration) registry.

Web services are composed by orchestration [40] to produce a workflow; orchestration is a form of coordination [85, 47], in which participants (web services) are separated from the coordination mechanism. In a workflow system, a number of actions (operation requests to web services) are performed in a sequence defined by the workflow. WS-BPEL (web services business process execution language), simply known as BPEL, is a well known industry specification that standardises orchestration [39, 37]. The program implementing orchestration represents the business process. We consider a simple online booking system Plan-a-Tour (PaT) shown in Figure 2.12.

We assume three web services WS1 (to book an airline ticket), WS2 (to book a hotel room) and WS3 (to book a taxi for airport pickup) available online on web servers
S1, S2 and S3, respectively. In order to plan a tour (e.g. to attend a two days international conference), a customer has to use these three web services for desired bookings for the tour. For the PaT system, using BPEL language, a workflow is created by orchestration of the three available services. The workflow is then converted into a web service WS4 by creating a WSDL for the workflow; WS4 is hosted on web server S4. A client program can call WS4 to make the three bookings in sequence.

2.6 Chapter Summary

In this chapter, we have introduced the component-based development paradigm which is the background of our work. As composition is the essence of software component models, we presented four general categories of composition and have proposed a new taxonomy of software composition mechanisms for the three categories of components. Next, we briefly presented component composition in the well known representative models from the three categories of component models. We will be using the knowledge gained from these examples in the next chapter to analyse how to achieve incremental construction in CBD.

In the next chapter, we define incremental construction in CBD and present a study of achieving incremental construction based on the three categories of component models.
Chapter 3

Incremental Construction in CBD

In the context of achieving incremental construction in CBD, the purpose of this chapter is to present a study from the three categories of component models, as described in Section 2.3. In the beginning, incremental construction is defined with an emphasis on the behaviour containment. Next, three general system construction approaches are described in the following section to present our view of bottom-up construction approach. In the next section, before the study of incremental construction in the three categories of component models, the study outline is presented very briefly. By comparing the strengths and weaknesses of the three categories of component models with respect to fixed behaviour of components, we select the X-MAN component model to achieve incremental construction. Lastly, in this chapter, a brief review of related work is presented.

3.1 Incremental Construction

In this section, adopting a top-down approach, we redefine the notion of incremental construction for CBD with an emphasis on behaviour containment. First, with an abstract view of a system, the incremental construction process and a construction step in the process are defined. Next, we define components and system behaviour in details.

3.1.1 Incremental Construction Process

By incremental construction, as the name suggests, a system is constructed iteratively by incrementing an incomplete version of the system until the system is completed. In
the three categories of component models, a system or a composite is a collection of components composed by connections or coordinators, as described in Section 2.5. In a system, components constitute the computation part of the system and connections or coordinators constitute the communication part of the system. Hence, the system can be represented by a pair \((\text{Comp}, \text{Comm})\) of computation and communication. The behaviour of a system is based on the computation and communication both. A system’s behaviour increments if incrementing the system results in adding new computation and/or communication in the system. Therefore an increment can be a component, a connector (connection or coordinator) or a composite (of components and connectors). The behaviour of a system is a set of services exhibited by the system; a service’s behaviour is the execution of computation from one or more components and possible interactions between the components.

![Figure 3.1: Incremental construction process](image)

The construction process (Figure 3.1(a)) begins with an initial incomplete system \(S_0\) (a component or a composite). Next, the construction step adds an increment \(inc_0\) to the system \(S_0\) to create an incremented system \(S_1\) and so on. An incremental step, in the process of incremental construction, is shown in Figure 3.1(b); the relation between the two consecutive systems is expressed as \(B_{S_i} \subseteq B_{S_{i+1}}\). In this relation, \(B_{S_i}\) and \(B_{S_{i+1}}\) represent the behaviours of systems \(S_i\) and \(S_{i+1}\) respectively; the symbol ‘\(\subseteq\)’ represents the behaviour containment, i.e., the behaviour of the new system \((S_{i+1})\) contains the behaviour of the old system \((S_i)\).

The behaviour of a system is a set of services exhibited by the system. An increment \((inc_i)\) can be a component, a connector or a composite; hence, the new (or incremented) system also contains the behaviour of the increment added to the system. However, the exact structure of an increment and the precise nature of the behaviour of the increment is different for different categories of component models.
3.1. INCREMENTAL CONSTRUCTION

3.1.2 Component and System Behaviour

In an incremental construction step, the behaviour of the incremented system $S_{i+1}$ contains the behaviour of the previous system $S_i$. Therefore, in order to understand the behaviour containment, it's important to understand the behaviour of components and system. Hence, in this section, we further describe the behaviour of a system (from Section 3.1.1) in detail. Moreover, the behaviour of an individual component is also described.

By adding further details, the system pair ($\langle Comp, Comm \rangle$) from Section 3.1.1 can be represented as a tuple ($\langle Comp, C, D \rangle$) of computation ($Comp$), control ($C$) and data ($D$). In a system, control triggers computations (from one or more components), which are function or expression evaluations, and assignments, amongst other. Computations are performed on data in the system. An execution of a system’s service (from the set of services exhibited by the system) may invoke many computations from more than one component, as described in Section 3.1.1. Hence, for a service’s execution request (from the set of exhibited services), a system’s behaviour is the result of executing its computations (according to its control flow) on its data.

A system is a set of interacting components; components contain behaviour (computation and communication) and interactions define message (control/data) communications between the system components. As with system, the behaviour of a component can be defined as a set of (provided) services exhibited by the component, e.g. component $C$ in Figure 3.2(a). The control flow in the computation of a service of a component defines interactions on the component’s ports; this knowledge of interactions through ports is required to use the component in a system. For generality, considering sequential systems with a single control flow, a provided service can be represented as a sequence of request/response messages through the component ports. Hence, for a service’s execution request, a system’s behaviour is the result (generation of messages on ports) of executing its computations (according to its control flow) on its data.

In a system, a component interacts with other components by exchanging messages through its ports and an interaction on a port is a tuple ($\langle Req, Res \rangle$) of request and response messages. A component’s behaviour can be represented by a set of message sequences (Figure 3.2(a)); a message sequence represents a provided service. In a message sequence for a provided service, the service’s computation is invoked with the arrival of the first message (in the sequence) and rest of the messages (except the last message) are either emitted (request messages) or received (response messages).
through the component’s required ports defined within the invoked service’s computation. The generation of the last message (in the sequence) indicates the completion of the invoked service.

To define a component’s behaviour, the use of messages on the component ports is not a new concept. Interaction protocols in Wright [5], behaviour protocol in SOFA2 [25], gate (port) protocols in TrustMe [90] and RDSEFFs (Resource Demanding Service Effect Specification) in Palladio [20] component models use the concept of interactions on component ports.

In Figure 3.2(b), we consider component $A$ with two provided services $S_1$ and $S_2$ (with specific message sequences) and two required services ($S_3$ and $S_4$). Service $S_1$ evaluates a mathematical expression ($S_1(a, b) = a^2 + b^3$; where $a$ and $b$ represent numeric variables) and service $S_2$ simply doubles a numeric value ($S_2(a) = 2 \times a$). The component requires two services ($S_3$ and $S_4$); service $S_3$ is required to square a numeric value ($S_3(a) = a \times a$) and $S_4$ is required to cube a numeric value ($S_4(a) = a \times a \times a$). The behaviour of a component as a set of message sequences shows the dependency of a provided service on the required service(s). For component $A$, service $S_1$ has dependency upon two external services $S_3$ and $S_4$; service $S_2$ is independent of any external dependency.

In order to illustrate the behaviour of a system in terms of its provided services (or in terms of sequences of messages on component/system ports), we construct a system by composing components $A$ and $D$ (with one provided service $S_5$), as shown in Figure 3.2(c). In a system, assuming that a provided port can be connected to more than one required port and a required port can be connected to one provided port, all provided and unconnected required component ports are ports of the system.
As with a component, the behaviour of a system can be defined in terms of sequences of messages for all services provided by the system, as shown in Figure 3.2(c). The system behaviour is a set of three provided services (\(S_1, S_2\) and \(S_5\)) or message sequences for these services on system ports (shown as \(B'_S\)) or on system’s components’ ports (shown as \(B_S\)).

What is a system interface or behaviour? Is it simply a collection of services (or provided ports) of all components in the system? In the bottom-up development approach, a component’s behaviour (selected from a repository of pre-developed components) may not be required fully to be part of the system. For example, composing an EJB component or a web service as an increment to the current system, few out of many provided services (methods or procedures) may be included in the system’s behaviour. Selection of component services in a system is decided by the system developer.

### 3.1.3 Behaviour Containment

Containment of the current system’s behaviour in the incremented system’s behaviour is the core concept of incremental construction. Considering component A in Figure 3.2(b) as the initial system and the composite in Figure 3.2(c) as the incremented system, the behaviour of initial system is a set of two services (or sequences of messages) and the behaviour of the incremented system is a set of three services. The sequences of messages of service \(S_1\) from the two systems are not equivalent which implies that the behaviour of \(S_1\) in the two systems is different. However, the sequence of messages of service \(S_1\) of component A is contained in the sequence of messages of service \(S_1\) of the composite. This containment of sequence of messages shows that \(S_1\) of the incremented system is the composite (compound) service of service \(S_1\) of A and service \(S_5\) of D. Service \(S_2\) of the two systems have the same sequence of messages. Sequence of messages of both services of the initial system are contained in the behaviour of the incremented system. Hence, the behaviour of the incremented system (\(B_S\)) contains the behaviour of the previous system (\(B_A\)). The containment relationship between the two behaviours is expressed as ‘\(B_A \subseteq B_S\)’.

In order to further illustrate the behaviour containment, the current system from Figure 3.2(c) is composed with component E (an increment) to create a composite component F (the incremented system), as shown in Figure 3.3. The behaviour of the incremented system contains the behaviour of the previous system. Hence, according to the definition of incremental construction in Figure 3.1(b), the construction step in
the two examples (Figure 3.2(c) and Figure 3.3) is incremental.

\[
B_F = \{ S_1, S_2 \} = \{ (m_1, m_6), (m_7, m_8) \}
\]

\[
B_S \subseteq B_F
\]

Figure 3.3: A specific example of Behaviour containment

The incremented system’s behaviour contains the behaviour of the current system in two ways: (i) original services of the current system are exhibited by the new system (e.g. \( S_2 \) of \( S \) is delegated to \( F \) in Figure 3.3) and (ii) the incremented system offers new services by combining services of the current system and of the increment (e.g. by port connection between \( S_4 \) and \( S_6 \), \( S_1 \) of \( S \) is combined with \( S_6 \) of \( E \) to become compound service \( S_1 \) of \( F \) in Figure 3.3). Behaviour of a system can be represented as a set of provided services or as a set of message sequences on the system ports, as shown in Figure 3.3.

To illustrate the concept of behaviour containment, we have described the service containment at the level of messages on ports by considering services as sequences of messages. Next, we consider an example (Figure 3.4) of a simple ATM (Automated Teller Machine) system to show different possibilities of behaviour containments at the level of services.

Figure 3.4: Behaviour containment
3.1. INCREMENTAL CONSTRUCTION

In this example, without showing the structure of the system or increment and how the increment \(inc_0\) (exhibits one service to deposit cash) is added to the current system \(S_0\) (exhibits one service to withdraw cash), behaviour containment property of incremental construction can be satisfied in five cases, as shown in Figure 3.4.

In case 1, \(S_1\) contains the atomic services of \(S_0\) and \(inc_0\), and a combined service of the two atomic services from \(S_0\) and \(inc_0\). In the combined service (\(transfer\)), output of the withdraw service is passed as input to the deposit service. The composite of case 4 collects the services of \(S_0\) and \(inc_0\). The containment relationship between the two behaviours \(B_{S_0} \subseteq B_{S_1}\) is true for all five cases. In three cases (case 1, case 2 and case 4), \(S_1\) contains services of \(S_0\) by set containment; such behaviour containment may not be needed in each incremental step of the construction process.

In incremental construction, one advantage of behaviour containment is that test cases of a current system \(S_i\) can be useful for testing the incremented system \(S_{i+1}\). In the ATM example, test cases for system \(S_0\) are also applicable on system \(S_1\) in three cases (case 1, case 2 and case 4). This implies that the test cases for \(S_0\) may not required to be verified for \(S_1\). However, for case 3 and case 5, the behaviour of \(S_0\) is contained by service combination. The test cases of \(S_0\) are not directly applicable to the incremented system \(S_1\) but they can be integrated into the test cases for \(S_1\). For example, testing the \(transfer\) service of \(S_1\) indirectly tests the \(withdraw\) service test case of \(S_0\).

3.1.4 Incremental Composition

In a component model, a system may be incremented by using composition mechanisms in the component model. For system construction, a composition mechanism defines interactions between the composed components, and creates a composite (system) that is a set of interacting components, as described in Section 2.4.2. In a composite, composed components represent sub-systems, such that behaviours of the composed components constitute the behaviour of the composite. A composite can further be composed with more components and/or composites to create an even larger composite.

Corresponding a composition mechanism with the definition of incremental construction in Figure 3.1(b), we assume that a composition mechanism composes two components \(C_1\) and \(C_2\) to create a composite. Component \(C_1\) corresponds to \(S_0\) and component \(C_2\) corresponds to \(inc_0\), the composition mechanism is referred to as incremental composition if the composite’s behaviour contains the behaviour of \(C_1\). Such a
composition mechanism supports incremental construction.

Hence, composing components iteratively by using a composition mechanism is an incremental construction approach if the composition of \( S_i \) (component/composite) with an increment \( inc_i \) yields a new composite \( S_{i+1} = S_i + inc_i \), such that the behaviour of \( S_{i+1} \) contains the behaviour of \( S_i \). The composition mechanism used in an incremental construction approach thus performs incremental composition.

### 3.2 System Construction Approaches

In our view, incremental construction is intrinsically bottom-up, and would therefore be easier to achieve by means of component-based development, which is also bottom-up. In this section, to clarify bottom-up approach, we describe three system construction approaches. The purpose of this section is to clarify that the bottom-up construction approach holds even if a required component is not available in the repository of developed components during system construction.

Garlan and Shaw [46] define a system’s architecture as a collection of components (data stores and/or behavioural units represented as boxes) and connectors (descriptions of the interactions between components represented as arrows or lines). Nierstrasz and Meijler [81] refer software architecture as a description of component composition to form a specific system. The top-level structure of a computer program (or software) is referred to as its architecture. This structure itself can be a structure of further sub-structures. Primarily, there are two system construction approaches, top-down and bottom-up. A third approach is referred to as the mixed approach.

<table>
<thead>
<tr>
<th>Top-down Approach</th>
<th>Mixed Approach</th>
<th>Repository</th>
<th>Bottom-up Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Top-down Approach" /></td>
<td><img src="image2" alt="Mixed Approach" /></td>
<td><img src="image3" alt="Repository" /></td>
<td><img src="image4" alt="Bottom-up Approach" /></td>
</tr>
</tbody>
</table>

![Figure 3.5: System construction approaches](image5)

The top-down construction approach starts with the top-level abstract system architecture. The complex and large systems are developed by following the divide and
3.3. THE OUTLINE OF THE STUDY OF INCREMENTAL CONSTRUCTION IN CBD

In order to be able to construct component-based systems incrementally, we need to have a component model that supports incremental construction. To explore the ability to achieve incremental construction, we study component models from the three categories in which components are objects, architectural units or encapsulated components. In this section, we present the broad outline of the study. In this study, as incremental construction is bottom-up, our underlying assumption is that all components with functional behaviour are available for system construction. An unavailable component can be developed and added into the repository.
3.3.1 Ways to Increment a System

For system construction, using a component model, a system can be constructed incrementally by adding increments bit by bit. In this study, to achieve incremental construction, mechanisms to add increments to a system are investigated in component models. For instance, composing an architectural unit with a system by port connection (Figure 3.2(c)) is an example of adding increment to the system, as described in Section 3.1.4. In some component models, increments can also be added by programming (or by refactoring) and by adapting.

Another way to increment an existing system seems to be by substituting an existing component with another component (with the signature based compatible ports). However, substituting a component by another component with the compatible ports may raise port tracing issue (sequence of messages of provided services) [90]. Moreover, non-functional properties of the services offered by the two components (developed by different vendors) may be different.

3.3.2 Behaviour Containment

In this study of the three categories of component models, to achieve incremental construction, the focus would be on the behaviour containment. In the rest of the thesis, the behaviour containment of the current system in the behaviour of the incremented system is investigated at the service level, as described in Section 3.1.3. However, if necessary, the behaviour containment can be investigated at the messages level, as used in Section 3.1.2.

3.3.3 Fixed versus Not-fixed Behaviour of Components

In CBD, by following the bottom-up construction approach, the initial system and every consecutive system in the construction process have fixed behaviour if components have fixed behaviour. The notion of “fixed behaviour” was proposed jointly by myself and two other colleagues Keng-Yap Ng and Kung-Kiu Lau in [62]; fixed behaviour means that a component’s behaviour with respect to control and computation is fixed. By adding such an increment, the system developer can verify (by executing) the system’s behaviour. In contrary, after adding a component with not-fixed behaviour, the developer has to take further actions or to add more increments to ensure that the composite has fixed behaviour. In the worst case, using component models with components with not-fixed behaviour, a system might have fixed behaviour when the system
is complete.

A component with fixed behaviour does not mean that the component is active by having its own control. For example, in the ProCom component model [24], ProSave components have fixed behaviour; behaviour of a component is a set of services. A service (computation) relates an input port group with a set of output port groups. A group of ports has one trigger port and one or more data ports. A service executes its computation by receiving control and data on an input port group and passes control and data on output port groups. A composite of two components $f$ and $g$ each with one service is shown in Figure 3.6. Component $f$ cubes a number and component $g$ squares a number. Components are connected by port connections. A connection (an arrowed line) between two ports simply transfers data or control from an out port of a component to an in port of another component.

![Figure 3.6: Components with fixed behaviour](image)

### 3.3.4 Construction of an Expression Calculator

In order to study the construction process in the three categories of component models, we consider a toy example that evaluates a mathematical expression ($c = a^2 + b^3$; where $a$, $b$ and $c$ are numeric variables). For this system construction in the three categories of component models, we assume that each computational function (behaviour) is a provided service of a component (which may be calling other services in its code). The system construction starts with one component that represents the initial system $S_0$. This initial system is a component with minimal behaviour (at least one provided service).
3.4 Object-based Component Models

Components in object-based component models are objects developed in an object-oriented programming language, as described in Section 2.4. Hence, for system construction, composition mechanisms for objects from our taxonomy (shown in Figure 2.7) may be used to increment the system, as described in Section 3.1.4. In this section, we go through the composition mechanisms for objects (from Figure 2.7) to explore the possibility of using them to increment the behaviour of a system. We also analyse the possibility of incrementing a system by adding code.

3.4.1 Increment by Composition

Objects/classes are composed by containment (class nesting, object composition and object aggregation), by extension (inheritance) and by connection (object delegation) mechanisms. For classes, containment and extension composition mechanisms construct a single class. In contrast, the composition of two objects by object delegation produces a composite of two objects connected by object delegation (message passing mechanism).

A system is a set of objects with at least one object with a system initialiser method (e.g. the main method). One can build a single class as a complete system; this is how systems are constructed in procedural-oriented development paradigms. Typically, in object-oriented development and CBD paradigms, a system is a set of objects connected by message passing mechanisms (e.g. object delegation). Nevertheless, for incremental construction, some composition mechanisms along with adding code can be used to add behaviour to the system.

The behaviour of a single object (the sys class) is the execution result of the object’s main method, as shown in Figure 3.7(a). The system’s behaviour is to execute two methods, the constructor method (special method to initialise class’s attribute variables) and the run method in sequence.

In Java, a class can have inner classes which are instantiated and used as normal objects. Composing inner classes to the sys class means adding the code of inner classes inside the code of the sys class, as shown in Figure 3.7(b). This is the class nesting composition mechanism (shown in Figure 2.7). However, composing inner classes into the sys class by containment does not add any behaviour to the system. In order to achieve an increment in the system’s behaviour by the composed inner classes, the inner classes must be instantiated as data members of the sys class and then their
methods must be called within the behaviour of the sys class.

Object composition/aggregation are two composition mechanisms to compose objects inside the definition of the composing class. Now we assume that we compose two objects by object composition/aggregation to the sys class; however, doing so alone does not add any new behaviour to the system. In order to achieve an increment in the system’s behaviour from the composed objects by object composition/aggregation, the behaviour of composed objects must be called inside the system’s behaviour.

Multiple class inheritance is a composition mechanism to compose two or more classes (shown in Figure 2.7). The sys class can be incremented by multiple inheritance (without overriding), as shown in Figure 3.8(a). The sys class will be incremented by the public members of the two parent classes C1 and C2. In invasive software composition [9], ‘single inheritance’ is also a composition mechanism because two independently developed fragment boxes (class boxes corresponding to classes in object-oriented development) can be composed by ‘single inheritance’, as shown in Figure 3.8(b) & (c). The inherited class is a pair of inherited part and incremented part [98]. The single inheritance shown in Figure 3.8(b) is incremental if C1 does not override any method of sys class. As an alternate, the single inheritance shown in Figure 3.8(c) is incremental for not destroying (or overriding) the system’s behaviour. Class inheritance extends a child class with the members of the parent class(es); however, the system behaviour is not incremented by class inheritance alone. In order to achieve an increment in the system’s behaviour from the extended members of the parent classes, the extended member methods must be called from inside the system’s behaviour.

Object delegation is a composition mechanism which composes two objects by one calling a method of another object. In Figure 3.9, we increment the behaviour of the
system by adding a method call (shown in bold) to object C1 inside the behaviour of the system. Object delegation is a message passing mechanism between the caller object and the called object. To increment a system’s behaviour, object delegation is an example of incremental composition if the method call is added in the behaviour of the system.

```
public class sys {
    public void m1() {
        ...
    }
    public void run() {
        m1(); C1.m2();
    }
    public static void main(String args[]) {
        u.run();
    }
}
```

An aspect [56] defines a crosscutting concern for some base class that can be woven with the base class to change the behaviour of the class. In an aspect, advices represent behaviour that can be added at various join points (locations of composing class) specified in pointcuts (that identify matching joinpoints). Weaving between a class and aspects is performed by an aspect weaving mechanism, which is implemented in a special language processor that weaves advices into a base class. Aspects can also extend the base class by adding new methods known as inter-type declarations. In Figure 3.10, we show how an aspect after weaving increments the behaviour of the sys class. The aspect adds the effects of object delegations to object C1 in the method m1 of the sys class. To increment a system’s behaviour, aspect weaving is an example of incremental composition if an aspect is weaved in the behaviour of the system.
In mixin-class inheritance (Figure 2.7) a mixin is composed with a class to increment the class. A mixin is referred to as an abstract subclass [22] that represents a fragmented program unit, which is not meant to stand alone but to be composed with a class in object-oriented languages. As with a class, a mixin is a program unit that represents a specific functionality to be inherited at subclass level(s) in a class hierarchy (but not at the root level). In the mixin-class inheritance, a method from the mixin overrides the method with the same signature in the class. As with class inheritance, mixin-class inheritance without overriding adds behaviour to the class but not to the system. Mixin-class inheritance with overriding overrides the behaviour of the system; this mechanism is not behaviour preserving and therefore not incremental.

Trait is a unit of reuse [36] which provides a set of services (shown with lollipop symbol) and may also require a set of services (shown with arrowed symbol). Provide services are a set of methods to undertake some useful computation. Require services are also a set of methods but these do not have any computational task but are used as parameters to the provide services. Traits are program units in Squeak, open-source Smalltalk-80 dialect [21]. Traits can be composed with a class by trait-class composition.
In Figure 3.11, we show a composition of a TDisplay trait with the sys class. In trait based development, composing a trait with a class is the last composition step; therefore the class must provide all the require services of the traits. The sys class is shown with a provide method `getx` to satisfy the require service of the TDisplay trait. We also show an attribute ‘x’ in the sys class which is needed for the print service of the trait. Composing a system class with traits by trait-class composition does not increment the behaviour of the system. To increment the system’s behaviour, the print method (newly added behaviour to the sys class) must be called within the behaviour of the system (e.g. the main method of the sys class).

### 3.4.2 Increment by Adding Code

We now show an increment based on code for a repetition construct, as shown in Figure 3.12. In the example shown, a repetition construct (for-loop) is added in the system behaviour. In this increment, the behaviour of the system is incremented by adding a loop to repeat calling a local method. As with repetition construct addition, an increment to represent a select construct can be added to the system behaviour to execute some parts of the behaviour subject to the selection criteria.

![Figure 3.12: Increment by adding repetition construct](image)

The system’s behaviour can be increased by adding code (for the repetition/selection constructs) to a method that is being called within the system behaviour. In case the refined methods are not part of the system’s behaviour, the method calls to the refined methods may be added to the system behaviour.

### 3.4.3 Incremental Construction of The Expression Calculator

In truly CBD approaches, system construction is a bottom-up process, i.e. starting system construction from pre-built system-independent components [52, 29]. Hence, we
start constructing the expression calculator with one component \textit{calObj} \((S_0)\) which has one method \(m3\) with minimum functionality (to add two numbers). In the next step, \(inc_0\) is added to the system by aspect weaving, as described in Section 3.4.1. Component \(sObj\) is composed with \textit{calObj} by object delegation, as shown in Figure 3.13(b). Let us assume that an aspect \(A\) has an advice with a method call statement for method \(m1\) of \(sObj\). Aspect \(A\) can be weaved with \textit{calObj} to extend the code of \textit{calObj}'s method \(m3\). This increment is added by composition, as described in Section 3.4.1. The incremented system \(S_1\) (Figure 3.13(b)) adds the squared value of the first number with the second number. The behaviour of \(S_1\) contains the behaviour of \(S_0\) as method \(m3\) of \(S_1\) contains method \(m3\) of \(S_0\), as described in Section 3.1.3. Method \(m3\) of \(S_1\) also contains method \(m1\) of \(inc_0\).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{system.png}
\caption{A system to evaluate a mathematical expression}
\end{figure}

In the next step, we increment the system by inheritance and add a local method call (again by aspect weaving). In this increment \((inc_1)\), we first extend \(sObj\) by class inheritance with \textit{cObj}, and then we extend \textit{calObj}, as shown in Figure 3.13(c). This is an example of adding an increment by composition, as described in Section 3.4.1. The incremented system \(S_2\) (Figure 3.13(c)) now adds the squared value of the first number with the cubed value of the second number. The behaviour of \(S_2\) contains the behaviour of \(S_1\) as method \(m3\) of \(S_2\) contains method \(m3\) of \(S_1\), as described in Section 3.1.3. Method \(m3\) of \(S_2\) also contains method \(m2\) of \(inc_1\).

Another way of construction is to create a new coordination class \(S\) and construct the system by programming. In the system class \(S\), first a call to method \(m1\) of \(sObj\) is made to get the squared value of a number. Next, a method call to \(m2\) of \(cObj\) is made to get the cubed value of a number. Lastly, a method call to method \(m3\) of \textit{calObj} is made to get the addition of the squared and the cubed values. The system class \(S\) is coordinating with the three components.
Based on the components, there are many different ways to construct a system from pre-built system-independent components.

### 3.4.4 Discussion and Evaluation

Components in object-based component models are developed in an object-oriented language. Composition mechanisms supported by the language and other programming constructs to add behaviour to objects can be used for system construction.

With the help of simple examples, we have shown that two composition mechanisms for objects, object delegation and aspect weaving, can directly add behaviour to the system. As these composition mechanisms support incremental construction, we refer to these as incremental composition. In contrast, in other composition mechanisms (class nesting, object composition/aggregation, class inheritance, mixin-class inheritance and trait-class composition), the added behaviour to the incremented class does not increment the system behaviour. In order to use these composition mechanisms as ways to increment the system behaviour, after composition, we need to add method calls to make the newly added behaviour be part of the system behaviour.

Other than composition mechanisms, behaviour can be added in three different ways: (i) to make methods of existing system classes be part of the system behaviour by method calls, (ii) to repeat system behaviour by adding a repetition construct, and (iii) to choose from alternate system behaviours by adding a selection construct.

In object-based component models, a system’s behaviour can be incremented by adding code either for object delegation or for repetition/selection of some behaviour. For system construction, an object-based component model can offer the flexibility of incrementing the system by adding code without destroying existing behaviour. However, this leads to low level program construction.

### 3.5 Architecture Description Languages

In the survey of software component models [72], components in many component models are architectural units (Figure 2.1). An architectural unit is a computational program unit with in/out communication channels (ports). Composition in the mainstream ADLs do not change the internal computation code and in turn the behaviour of the composed components. Conversely, the invasive software composition (ISC) changes the code of the composed components. We study incremental construction in
3.5. ARCHITECTURE DESCRIPTION LANGUAGES

the invasive ADL separately from mainstream ADLs.

3.5.1 Mainstream ADLs

Based on port types, architectural units can be categorised into three types: (i) architecture units with data ports, (ii) architecture units with procedure (service) ports and (iii) architecture units with event ports. Architectural units in the three categories are composed by port connection that define the message passing mechanism/policy between the connected/composed components. Based on the message passing mechanism and message anatomy, a connection can be categorised into three basic connector types [45]: (i) pipe (one-way data communication), (ii) procedure-call (two-way message communication) and (iii) event-broadcast (one-way message communication).

Composing a specific type of architectural unit by means of a specific type of connector is referred to as a composition or architectural style [44, 46]. The composition of architectural units with three types of connectors (pipe, procedure-call and event-broadcast) creates three composition styles: (i) pipe-and-filter style, (ii) client-server style, and (iii) publish-subscribe style. Data flow in one direction in Pipe-and-filter style. In client-server style, messages flow in both directions between composed components. Publish-subscribe style supports flow of messages in one direction between the composed components. A composition style with all three types of components and connectors is a hybrid style. These three styles are referred to as interaction styles in [30].

Component models support different architectural styles in different ways; for example, filter components in [4] have data (stream) ports and filter components in ProSave component model [24] have control ports. In this section, in order to avoid dissimilar features amongst component models with the same architectural style, we define three basic component models and analyse their support for incremental construction. The composition mechanism in these three component models is port connection and unconnected ports of the composed components are ports of the composite.

The Basic Pipe-And-Filter Component Model

In CBD, for pipe-and-filter architectural style [46], a filter is an independent component with one or more input and output data ports; filter components read data from input ports, transform data and write data to the output ports. A pipe transfers data from the sender to the receiver ports. Pipe-and-filter architecture is formally defined
in [4] and the ProSave component model (in the ProCom component model [92]) is based on a pipe-and-filter paradigm [24].

In the basic pipe-and-filter component model (basic-pnf), a component is an architectural unit with many in-/out-ports (Figure 3.14(a)). A component with at least one in-port and one out-port is a legal filter component in basic-pnf. Ports in filter components are channels for data communication. A data type is associated with each port that represents the type of data allowed to be communicated through the port.

Behaviour of a filter component is a set of functions and a function is a relation between a non-empty sub-set of in-ports and one out-port (Figure 3.14(b) shows behaviour of a specific component $K$). In order to keep the model simple, we assume that an in-port is related to one out-port. The behaviour of a component can also be represented as a set of out-ports; where an out-port is simply a function of in-ports, as shown in Figure 3.14(c) for $K$.

The filter component, on data arrival through in-ports, evaluates a function and produces data on one out-port. Figure 3.14(c) shows the execution semantics of a function of a component. In run-time, as soon as the data is available on in-ports, the related function is evaluated in three steps: (i) read in-ports atomically, (ii) compute the computation and (iii) write data on an out-port. Ports are read destruct, which means that the data on ports will be destroyed once read.

![Figure 3.14: A filter component and a composite](image)

In basic-pnf, filter components are composed by port connections; a port connection is referred to as a pipe that transfers data from one end (the sender port) to the other end (the receiving port). A pipe connects two non-similar (one in-port and one
out-port) ports by matching data types associated with the ports. The port of a filter component can be connected to one pipe. Composing two or more filters with pipes produces a composite filter (Figure 3.14(d)). All unconnected ports are the ports of the composite.

In Figure 3.14(d), considering component $A$ as initial system $S_0$ and component $C$ as increment $inc_0$; $inc_0$ is composed with $S_0$ by port connection and the composite $AC$ is the incremented system $S_1$. A function $f_5$ (to represent the composite function) relates the composite’s out-port ‘h’ with the composite’s in-port ‘a’. Corresponding with the definition of incremental construction from Section 3.1.1, behaviour of filters or the system is a set of functions. Function $f_5$ of $S_1$ is a compound function of $f_1$ of $S_0$ and $f_3$ of $inc_0$; $f_5$ contains $f_1$, as described in Section 3.1.3. The relation of behaviour containment between the two systems holds ($B_A \subseteq B_{AC}$), as described in Section 3.1.3; hence, basic-pnf component model supports incremental construction.

In order to illustrate incremental construction in basic-pnf, we use Modelica [42] to construct a system to evaluate a mathematical expression ($c = a^2 + b^3$; where $a$, $b$ and $c$ are numeric variables). Modelica is an object-oriented programming language used for modelling and simulation purposes. In Modelica, a block (an architectural unit) can have typed data ports (e.g. integer ports, real ports etc.). We develop three filter components (with one function each): (i) $calc$ to add two integer values, (ii) $sCom$ to square an integer value and (iii) $cCom$ to cube an integer value. The system is constructed in two steps, as shown in Figure 3.15.

In the first step, two components $calc1$ ($S_0$) and $square1$ ($inc_0$) are composed by

\[
B_{S_0} \subseteq B_{S_1}
\]

and in the second step, two components $calc1.f1(square1.f1(square1.a),calc1.b)$ and $cube1.f1(cube1.f1(cube1.a))$ are composed by

\[
B_{S_1} \subseteq B_{S_2}
\]

Figure 3.15: A system to evaluate a mathematical expression
port connection. In the second step, the composite ($S_1$ from Figure 3.15(a)) is composed with the cube component $\text{cube1 (inc}_1\text{)}$, as shown in Figure 3.15(b). The behaviour of both composites $S_1$ and $S_2$ is a composite function. After each incremental step, the behaviour of the incremented system $B_{S_{i+1}}$ contains the behaviour of the previous system $B_{S_i}$, as described in Section 3.1.3.

### The Basic Client-Server Component Model

Many component models (e.g. ACME, SOFA and UML) support client-server composition style. In this style, two components communicate by passing messages through their ports, as shown in Figure 2.9. The execution of the caller service (from client component) is paused by making a call and resumed on receiving the response from the called component.

In our basic client-server component model (basic-cs), a component is an architectural unit with many required ports (r-ports) and provided ports (p-ports), as shown in Figure 3.16(a).

![Figure 3.16: A component and a composite](image)

In basic-cs, a component with at least one p-port and zero or more r-ports is a legal component. A port represents a message communication channel between a component and its environment. A signature (of a procedure/service) is associated with each port that represents the type of message allowed to be communicated through the port. A component’s behaviour is a set of services and a service is a relation between a subset of r-ports and one p-port (Figure 3.16(b) shows behaviour of a specific component.
K). For a component, an interaction (sending a request message and receiving a response message) on a r-port is initiated from within the computation code of a service in the component.

In basic-cs, a r-port of a component may be related to one or many p-ports and a p-port may be related to zero or more r-ports, as shown in Figure 3.16(b). A p-port not related to any r-port represents that there is no external service request initiated within the computation associated with the p-port. The behaviour of a component can also be represented as a set of p-ports; a p-port is simply a function of related r-ports, as shown in Figure 3.16(c) for K.

In basic-cs, during execution, only one service of a component may be executed at a time. Figure 3.16(c) shows the execution semantics of a service of a component; a service interacts zero or more times on its r-ports. The execution of a service with associated r-ports is paused by requesting external services on its r-ports. The paused service resumes its execution after receiving a response message (control/data) from the requested service. In contrast to functions in a filter component (from basic-pnf), r-ports associated with a service are not data values but represent external service interactions initiated from within the code of the service. An interaction on these r-ports may not return a data value to the caller service.

In basic-cs, components are composed by port connection. A connection simply transfers a request message from r-port to the connected p-port and returns a response message from the p-port to the connected r-port. Two ports may be connected by matching the associated service signatures with the ports. For simplicity, we consider only one connection for a port. A composite of two components is shown in Figure 3.16(d); all unconnected ports are composite’s ports and the behaviour of the composite is a set of services (or unconnected p-ports).

In Figure 3.16(d), considering component A as initial system $S_0$ and component C as increment $inc_0$; $inc_0$ is composed with $S_0$ by port connection and the composite $AC$ is the incremented system $S_1$. A service s5 (to represent the compound service) relates the composite’s p-port ‘h’ with the composite’s r-port ‘a’; service s5 of $S_1$ contains service s1 of $S_0$ and service s3 of $inc_0$, as described in Section 3.1.3. As the relation of behaviour containment between the two systems holds ($B_A \subseteq B_{AC}$), basic-cs component model supports incremental construction.

In order to illustrate incremental construction, we consider constructing a system to evaluate a mathematical expression ($c = a^2 + b^3$; where a, b and c are numeric variables). We develop three components Calc (to add two numbers), $sCom$ (to square
a number) and \(cCom\) (to cube a number) in ArchJava [3, 2]; the desired system is constructed in two steps, as shown in Figure 3.17.

![Diagram of system to evaluate a mathematical expression](image)

Figure 3.17: A system to evaluate a mathematical expression

In the first step, two components \(Calc1\) (\(S_0\)) and \(sCom1\) (\(inc_0\)) are composed by port connection. In the second step, the composite (\(S_1\) shown in Figure 3.17(a)) is composed with \(cCom1\) (\(inc_1\)). The result of this composition is a composite \(S_2\) (Figure 3.17(b)). After each incremental step, the behaviour of the incremented system \(B_{S_{i+1}}\) contains the behaviour of the previous system \(B_{S_i}\), as described in Section 3.1.3.

### The Basic Publish-Subscribe Component Model

In the publish-subscribe architectural style, messages flow from one publisher component to many subscriber components. Publish-subscribe style is suitable to construct a system where many different components have to perform their specific computations subject to an event in one common component. In CBD, many component models (e.g. ACME, C2 [99] and SOFA) support this style.

In the basic publish-subscribe component model (basic-ps), a component is an architectural unit with zero or more publisher ports (pub-ports) and with one or more subscriber ports (sub-ports) (Figure 3.18(a)). Ports in components are channels for message (control/data) communications. A message signature is associated with each port that represents the type of message allowed to be communicated through the port. Behaviour of a component is a set of services and a service is a relation between a subset of pub-ports and one sub-port (Figure 3.18(b) shows behaviour of a specific component \(K\)). A sub-port only allows a message to pass into the component and a pub-port only allows a message to go out of the component.

In basic-ps, for a component, one pub-port may be related to many sub-ports, as shown in Figure 3.18(b) for \(K\). A component may have services in which no event is
raised within the service. A service signature is associated with a port. Behaviour of a component can also be represented as a set of sub-ports; sub-ports are simply functions of pub-ports, as shown in Figure 3.18(c) for $K$. Components in basic-ps have their own control. Figure 3.18(c) shows the execution semantics of a service of a component. A service executes by receiving notification on the associated sub-port and raises event(s) on associated pub-port(s).

In basic-ps, components are composed by port connection between matching ports, as shown in Figure 3.18(d); the result of composition is a composite. A port connection represents an event broadcast link between the event source port and the event sink port of the connected components. A connection simply transfers an event message from the pub-port of a component to the connected sub-port of the other component. In basic-ps, for simplicity, we consider that a sub-port may be connected to one pub-port at a time and a pub-port may be connected to zero or more sub-ports. All unconnected ports of the composed components are the respective ports of the composite.

In Figure 3.18(d), considering component $A$ as initial system $S_0$ and component $C$ as increment $inc_0$; $inc_0$ is composed with $S_0$ by port connection and the composite $AC$ is the incremented system $S_1$. Services $s_4$ and $s_5$ relate the composite’s sub-ports ‘b’ and ‘c’ with the composite’s pub-port ‘d’. Service $s_4$ of $S_1$ contains $s_1$ and $s_3$; similarly, service $s_5$ of $S_1$ contains $s_2$ and $s_3$, as described in Section 3.1.3. As the relation of behaviour containment between the two systems holds ($B_A \subseteq B_{AC}$), basic-ps component model supports incremental construction.

In order to illustrate incremental construction, we consider constructing a system...
to evaluate a mathematical expression \( c = a^2 + b^3 \); where \( a, b \) and \( c \) are numeric variables). We develop three components \( \text{calc}, \text{sCom} \) and \( \text{cCom} \) in ArchJava. The \text{op} service of \text{calc} accepts two numbers (of type integer) as arguments and broadcasts on its ports ‘b’ and ‘c’. Service \text{getSq} receives result on port ‘e’ and raises an event on port ‘d’ if port ‘f’ has been notified. Service \text{getCu} receives result on port ‘f’ and raises an event on port ‘d’ if port ‘e’ has been notified. Component \text{sCom} offers service \text{sq} to square a number and to broadcast the result on port ‘h’. Similarly a component \text{cCom} offers service \text{cu} to cube a number and to broadcast the result on port ‘h’.

In the first step, two components \text{calc1} (\( S_0 \)) and \text{sCom1} (\( \text{inc}_0 \)) are composed by port connection. In the second step, the composite (\( S_1 \) shown in Figure 3.19(a)) is composed with \text{cCom1} (\( \text{inc}_1 \)). The result of this composition is a composite \( S_2 \) (Figure 3.19(b)). After each incremental step, the behaviour of the incremented system \( B_{S_{i+1}} \) contains the behaviour of the previous system \( B_{S_i} \), as described in Section 3.1.3.

![Figure 3.19: A system to evaluate a mathematical expression](image)

**Special Cases of Incremental Construction**

For constructing a system in the three styles of ADLs, the only way to add an increment is by port connection. With the assumption that all components with functional behaviour are available for system construction, during incremental construction, we may face situations when an increment component cannot be connected to the current system. In this section, considering generic architectural units, we further discuss the two basic cases of composing two architectural units.

A typical component in ADLs has required ports; however, components in ADLs can have provided ports only. In the first case, considering any two components \( A \) and \( B \) with generic provided ports only (Figure 3.20(a)), a special component is needed
just to compose these two components.

In Figure 3.20, we assume that an incremental construction process starts with component \( A \) as the initial system \( S_0 \). In the next incremental step, component \( B \) is selected as an increment \( \text{inc}_0 \); component \( B \) cannot be added directly to the current system. In order to be able to add component \( B \) to the current system, component \( C \) is created just to add component \( B \) as an increment to the current system, as shown in Figure 3.20(b). The component \( C \) can coordinate communications between components \( A \) and \( B \). Similarly, just to compose, a component can be created to forward the ports of the two components, as shown in Figure 3.20(c). Generally, to compose any two components, ADLs do not provide such a special generic architectural unit (without its own computation). In order to compose such components, the ProCom component model introduces special architectural units (called connectors) to coordinate components [24].

In the second case, as shown in Figure 3.21, we consider two components \( A \) and \( B \) with no matching ports. In order to compose such components, an adapter component is needed that does not have any functional behaviour in the system. Generally, to compose any two components, ADLs do not provide such a special generic architectural unit (without its own computation). Components with incompatible ports in [15] are connected by using adapting filters; these filters adapt the type of message from one port to the acceptable type by the other port.
3.5.2 Invasive ADL

A component model with invasive composition is briefly described in Section 2.5.2; we refer to this component model as invasive ADL. This component model is using the concept of aspect weaving (described in Section 3.4.1) to change the code of a component. Hence, this component model is different than the mainstream ADLs. However, this model uses another different way to increment a system and that is by transforming the component code to extend and to connect with other components.

In order to illustrate incremental construction, we consider a system to evaluate a mathematical expression \( c = a^2 + b^3 \) where \( a, b \) and \( c \) are numeric variables). In order to show another way of adding increments to a system, we consider three components with provided ports only, as shown in Figure 3.22(a); however, components in invasive ADL can have required ports. Component \( \text{calc} \) has two declared and one implicit hook. In the first incremental step, component \( \text{calc1} (S_0) \) is composed with component \( \text{Sqr1} (\text{inc}_0) \) by using a composer program \( \text{comp1} \) (not shown). Composer \( \text{comp1} \) invades into one of the declared hooks to extend and connect \( \text{calc1} \) with \( \text{Sqr1} \), as shown in Figure 3.22(b); the invaded declared hook disappears from the component. In the next incremental step, the system from Figure 3.22(b) \( (S_1) \) is composed with \( \text{Cube1} \) by using another composer program \( \text{comp2} \). Composer \( \text{comp2} \) invades into \( \text{calc1} \)’s declared hook to extend and connect \( \text{calc1} \) from \( (S_1) \) with \( \text{Cube1} \), as shown in Figure 3.22(c); the invaded declared hook disappears from the component.

![Figure 3.22: A system to evaluate a mathematical expression](image-url)

Once invaded by the composer program, declared hooks disappear from the component’s composition interface. In contrast, implicit hooks do not disappear from the composition interface of the component and are still available for further transformations. The system of three connected components contains the transformed behaviour...
of \textit{calc1}. After each incremental step, the behaviour of the incremented system \((B_{S_{i+1}})\) contains the behaviour of the previous system \((B_{S_i})\), as described in Section 3.1.3. Using this component model, components with incompatible ports are not required to be composed by a third component (as shown in Figure 3.21), but such components can be transformed by a composer program for connection.

### 3.5.3 Discussion and Evaluation

In ADLs, systems are constructed by composition (port connection) of architectural units. Composition in mainstream ADLs does not change any of the existing behaviour of the architectural units. Conversely, composers in the ISC change the code of the composed components; this characteristic of invasion does not fit well with the general perception of a component as a black box entity that can be used via the specified interfaces. Moreover, code invasion has a tendency to destroy the existing behaviour of composed components.

In a system in these styles, an increment component with matching ports with a system component can be directly added to the system. This way of adding increments is referred to as adding increments by composition. In order to compose an increment component that does not have matching ports with the system, we have to create an architectural unit just to add the increment to the current system. In general, ADLs do not provide a generic architectural unit to add such components to a system.

### 3.6 Component Models with Encapsulated Component

In this category, from the current component models, we have included two component models in which components have provided ports (interfaces) only; such components are referred to as encapsulated components. In this section, we briefly investigate system construction with web services and with encapsulated components in X-MAN.

#### 3.6.1 Web Services

A web service hosted on a web server is available for system construction with a WSDL interface which represents the provided interface of the service. Web services are composed by programming a BPEL process which coordinates control between the composed web services. Programming of a BPEL process is referred to as orchestration, as described in Section 2.5.3. This process can be converted into a web service
by writing WSDL interface for the process; such a web service is referred to as virtual composition in [49]. Behaviour of a web service is a set of its operations specified in its WSDL.

In order to illustrate incremental construction by web services, we consider constructing a system (Figure 3.23) to evaluate a mathematical expression \( c = a^2 + b^3 \); where \( a, b \) and \( c \) are numeric variables.

![Diagram of incremental construction](image)

**Figure 3.23: A system to evaluate a mathematical expression**

We start constructing the system with one web service \( S (S_0) \) which has one operation to square a number value. In the first incremental step (Figure 3.23(a)), we compose web service \( S \) with another web service \( C (inc_0, \text{with one operation to cube a number}) \) by a BPEL process; the BPEL process is then converted to a web service \( SC (S_1) \). The composite web service \( SC \) has one operation which accepts two numbers and returns a pair, the squared value of the first number and the cubed value of the second number. In the second incremental step (Figure 3.23(b)), we compose the composite \( SC \) with another web service \( A (inc_1, \text{with one operation to add two numbers}) \) by a BPEL process; the BPEL process passes the values returned by \( SC \) as input values to the operation of web service \( A \). The BPEL process is then converted to a web service \( SCA (S_2) \) with one operation to evaluate the mathematical expression for the desired system. In a composite, numeric labels next to the interaction arrows represent the order of their occurrence. After each incremental step, the behaviour of the incremented system \( (B_{S_{i+1}}) \) contains the behaviour of the previous system \( (B_{S_i}) \), as described in Section 3.1.3.
3.6. COMPONENT MODELS WITH ENCAPSULATED COMPONENT

3.6.2 The X-MAN Component Model

In X-MAN, components have one provided interface and components do not call other components’ services directly. Components are composed by exogenous composition connectors. The behaviour of a component is the set of services exhibited by its interface.

In order to illustrate incremental construction, we consider constructing a system to evaluate a mathematical expression ($c = a^2 + b^3$; where $a$, $b$ and $c$ are numeric variables). In the component repository, there are three concerned atomic components $Calc$, $sCom$ and $cCom$: (i) $Calc$ has one service to add two numbers, (ii) $sCom$ has one service to square a number, and (iii) $cCom$ has one service to cube a number. By using instances of these components, we compose the system in two constructional steps, as shown in Figure 3.24.

![Figure 3.24: A system to evaluate a mathematical expression](image)

In the first step, we compose component $sCom1$ ($S_0$) with component $cCom1$ ($inc_0$) by a sequencer connector (SEQ1). The composite $S_1$ (Figure 3.24(a)) has one service which accepts two numbers as inputs and returns a tuple of two numbers (the squared value of the first input value and the cubed value of the second input value). In the second step, we add $Calc1$ ($inc_1$) with the current system $S_1$ by means of a pipe connector (PIPE1). The composite $S_2$ (Figure 3.24(b)) has one service which accepts two numbers and returns the result of addition of the squared value of the first input and the cubed value of the second input. In this composite, the computed results from SEQ1 are passed as input values to $Calc1$ by PIPE1. After each incremental step, the behaviour of the incremented system ($B_{S_{i+1}}$) contains the behaviour of the previous system ($B_{S_i}$), as described in Section 3.1.3.
3.6.3 Discussion and Evaluation

A web service is a component running (or ready to be executed) on a web server with a WSDL interface. For incremental construction, the developer needs to ensure that the incremented system contains the behaviour of the previous system.

In X-MAN, components are composed by exogenous composition connectors; composing this way creates a composite whose behaviour contains the behaviours of the composed components. Such a composition is referred to as incremental composition, as described in Section 3.1.4.

The composition of web services is a (BPEL) process which has to be converted to a web service for further composition. Moreover, as there is no notion of pre-defined connectors for system construction, a BPEL process is coded from scratch. Conversely, the composition of X-MAN components creates a composite that can be composed further. X-MAN offers a set of pre-defined connectors which can be re-used in system construction. Hence, a system in X-MAN is constructed by adding components and connectors.

3.7 A Comparative Study of the Three Categories of Component Models

In this section, in order to select a suitable component model for incremental construction, we will draw on the knowledge of our study of the three categories of component models. The criterion for selecting a component model is based on the fixed behaviour of components, as mentioned in Section 3.3. In this Section, based on component’s behaviour, the studied component models are categorised into two groups. Next, using the flow of (control/data) messages through unspecified/specified ports, behaviours of objects and architectural units are described in details to justify the classification of component models.

3.7.1 Behaviour Based Comparison

In this study, with respect to component behaviour, component models from the three categories can be grouped into two categories, as shown in Figure 3.25. Using component models from the category of not-fixed behaviour for incremental construction, a system’s behaviour may be fixed once the system is completed. In contrast, using
3.7. A COMPARATIVE STUDY OF THE THREE CATEGORIES OF COMPONENT MODELS

Component models from the category of fixed behaviour for incremental construction, the final system as well as the intermediate systems have fixed behaviour; functional behaviour of such systems can be verified.

<table>
<thead>
<tr>
<th>Component Behaviour</th>
<th>Component Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not-Fixed</td>
<td>Object-Based Component Models</td>
</tr>
<tr>
<td></td>
<td>Basic Client-Server ADL</td>
</tr>
<tr>
<td>Fixed</td>
<td>Basic Pipe-And-Filter ADL</td>
</tr>
<tr>
<td></td>
<td>Basic Publish-Subscribe ADL</td>
</tr>
<tr>
<td></td>
<td>Component Models with Encapsulated Components</td>
</tr>
</tbody>
</table>

Figure 3.25: Behaviour based categories

Components with Not-fixed Behaviour

A composite of two components $A$ and $B$ is shown in Figure 3.26; we use request and response messages for interactions between composed components, as described in Section 3.1.2. A message represents control and/or data flow. An interaction begins with a request message on a provided port and ends with the response message form the provided port.

Using object-based component models, to construct a system, objects can be implemented as encapsulated components, as shown in Figure 3.13(a). However, typically, objects in a system have external method calls in their methods; an object with external method calls do not have fixed behaviour with respect to control and computation. In Figure 3.26(a), the behaviour of component $A$ is a single method $m1$ which calls method $m2$ of component $B$. The behaviour of component $B$ is a set of two methods, $m2$ and $m3$; method $m3$ calls method $m4$ of some other component. During execution of methods $m1$ (of $A$) and $m3$ (of $B$), control leaves the component and returns back after executing an external method. The behaviour of component $A$ depends on the behaviour of component $B$. Similarly behaviour of component $B$ depends on the behaviour of a specific component. With respect to control and computation, behaviours of $A$ and $B$ are not fixed as including one of these components implies to have more components; hence, this leads the developer to add more components.
In the study of achieving incremental construction in object-based component models (from Section 3.4), a truly component-based construction may be achieved artificially if components are implemented as encapsulated components, as shown in Figure 3.13. However, achieving incremental construction by the weaving mechanism is limited as the code (e.g., external method calls) cannot be inserted everywhere in a component to achieve behavioural increment to the system.

In the basic client-server style described in Section 3.5.1, a typical architectural unit has required ports. However, architectural units can also be implemented as encapsulated components, as shown in Figure 3.20. For a typical architectural unit, behaviour of an architectural unit is not-fixed with respect to control and computation. Required ports of a component represent the dependencies of the component upon other components with matching ports. We transform the object specific composite from Figure 3.26(a) into a composite with architectural units, as shown in Figure 3.26(b). As with object specific composite, the behaviour of component A depends on the behaviour of component B. However, unlike component B in Figure 3.26(a), behaviour of B in Figure 3.26(b) depends on the behaviour of some other component (compatible for connection on B’s required port).

With the help of flow chart notations, computation in a component with not-fixed behaviour (or with dependencies upon other components’ computations) is shown in...
3.7. A COMPARATIVE STUDY OF THE THREE CATEGORIES OF COMPONENT MODELS

Figure 3.26(c). Computation of component A’s method or service halts by making request to a method or service of component B; A’s halted computation resumes after receiving response from B.

Components with Fixed Behaviour

In Section 3.5.1, architectural units in the basic pipe-and-filter and basic publish-subscribe styles have their own control. For a filter component, a computation is fixed to read data from the in-ports and to produce data on the out-ports; data on in-ports and data on out-port correspond to the request and response messages (from Section 3.1.3) respectively. Similarly, in the publish-subscribe style, computation of a component is fixed to listen to event notifications on the sub-ports (subscriber ports) and to raise events on the pub-ports (publisher ports). For the execution of a service, a notification on a sub-port corresponds to the request message and an event generation on a pub-port corresponds to the response message (from Section 3.1.3). In a component of these two styles, every computation runs to complete its computation once started; components have no external dependencies to complete their computations. Such components have fixed behaviour.

Encapsulated components do not have any required ports; therefore encapsulated components do not have any dependencies upon any other component. Component models with encapsulated components are placed in the category of fixed behaviour.

3.7.2 A Component Model for Incremental Construction

So which component model should we choose for incremental construction? The intention is to pick a component model which can help constructing a system incrementally (in many iterations) such that the system has fixed behaviour after each increment. The benefit of a system with fixed behaviour is that the system’s behaviour is verifiable after each increment without stubs (artificial components producing expected result values without real computation). Being able to verify a system with fixed behaviour reduces the extra overhead of stubs for subcomponents [11].

In object-based component models, components calling other components’ services are component type-bound. For example, a component A with a method call of component B. In a system, by adding component A implies that component B must also be added. In contrast, components in ADLs are not component type-bound. Therefore, in system construction, adding a component A (with required ports) implies that any
component with matching provided ports with component A’s required ports may be added. However, in both cases, adding a single component with dependencies upon other components (excluding the input/output devices or the components connected to the input/output devices) leaves the system with not-fixed behaviour. To have a system with fixed behaviour by using a component model from the ‘Not-Fixed’ category, the construction process may begin with a component without any dependencies (e.g. an encapsulated component). Then an increment is added to the system to create the incremented system such that the system has fixed behaviour. In the worst case, the system may be constructed in a single step (big-bang integration) if there is no encapsulated component for constructing a system.

In contrast, component models from the other category can be used to construct the system in the desired way. A system constructed in a pipe-and-filter component model have components which can generate data and pass data among themselves. Pipe-and-filter style systems are data driven systems. A system constructed by a pipe-and-filter component model has limited expressiveness because of having components with many ports and in turn many connections between the components. Similar is the case with component models with the publish-subscribe composition style, as shown in Figure 3.19. A component model with publish-subscribe composition style is more appropriate for building GUI (graphical user interface) software [99].

In the category of component models with fixed behaviour, we studied two component models with encapsulated components. There are many similarities between web services and X-MAN component models. These are general purpose component models which can deal with the data/control flows in a system. The system architecture in both these component models are similar to the tree structure, as shown on Figure 3.23 and Figure 3.24. In both these component models, the composition mechanism is control coordination (Figure 2.7). Despite aforementioned similarities, the two component models are fundamentally different.

Web services are composed by arbitrary BPEL code to generate a workflow (a coordinating process), as described in Section 2.5.3. A coordinating process may be turned into a web service for hierarchical composition. Generic coordinating processes are not defined as reusable programs which can be composed in general. In the incremental construction, the behaviour containment of the current system in the incremented system is manually ensured by the BPEL process developer.

In contrast, X-MAN components are composed by pre-built exogenous composition connectors to produce a composite which can be further composed. Exogenous
composition connectors define incremental composition for X-MAN components as the behaviour of the composite contains the behaviours of the composed components. Furthermore, X-MAN also defines other kinds of increments (e.g. adding adaptors), apart from components, that can be added to a system. Components and connectors in the conventional X-MAN component model are passive. Moreover, in an X-MAN system, computation and control can be clearly segregated (see Section 4.1.4); this may not be possible in a system constructed by web services as the BPEL processes can also have some computation code [11]. In order to construct systems incrementally, we will use the X-MAN component model.

### 3.8 A Review of Related Work

In the study of three categories and different composition styles of one category of component model, four out of five different kinds/styles of component model (Figure 3.25) are studied in general; the study of component models with encapsulated components is specific to web services and X-MAN.

For supporting development for/with reuse [70], CBD is a reuse-oriented approach, as described in Section 2.1. In CBD, there is a long list of component models; this is difficult to decide that which component models should be included in the related work. We select some widely known and representative component models from the three categories of component models. Moreover, we select Reo (a channel-based coordination model for component composition) [7] as a related work to the X-MAN component model.

In this section, for independent reading, the selected models are described briefly and then investigated for their resemblance with the five styles of component models (Figure 3.25) studied in this chapter. In the context of incremental construction, we further describe a selected model if the corresponding style from our study does not reflect the true capability of the model.

#### 3.8.1 JavaBeans

JavaBeans [76] is a software component model in which a component is a special Java class with properties, methods and events [72]. Two independent beans are composed by one (receiver/target bean) registering its interest in an event fired by another bean (sender/source bean). Listener methods in a bean correspond to the provided services
and event methods in the source beans correspond to the required services (in the sense that the call of an event listener method is initiated from the event source object) of the generic component (Figure 2.2(a)). However, unlike required services, a bean component does not require any listener to be registered to its event methods. Beans can be visual (e.g. juggler bean, GUI bean) or non-visual (e.g. a spell checker) components.

In Figure 3.27, using BeanBox (a builder tool in JavaBeans Development Kit (BDK[32])), a system is created in two steps. In this system, three bean components are composed by registering (shown as a line) the animator bean with two button beans. The animator bean has two listener methods to start and to stop the animation, along with many other public methods. A click on the start button starts the animation and a click of the stop button stops the animation. BeanBox generates an action listener object for each registration line, as shown in Figure 3.27(c).

Figure 3.27: Composition of JavaBeans

JavaBeans supports incremental construction and the system architecture resembles with the publish-subscribe style from Section 3.5.1. However, unlike the basic-ps, event/listener methods’ signatures are not required to be matched for connecting two beans and each method can be connected more than once. Development in JavaBeans falls in the event-driven development paradigm.

### 3.8.2 ProCom Component Model

In the ProCom component model, a system has two distinct but related layers of components: ProSys component layer and ProSave component layer. In the upper layer, ProSys components represent active sub-systems connected by message passing mechanism. In the lower layer, ProSave components represent the internal functional components of a ProSys component from the upper layer. ProSave components are passive
components and they communicate by passing data on their ports. The ProSave component model is based on a pipe-and-filter paradigm [24].

A composite $(S_i)$ of two components $f (S_0)$ and $g \ (inc_0)$ is shown in Figure 3.6; behaviour of the incremented system $(S_i)$ contains the behaviour of previous system $S_0$, as described in Section 3.1.3. Hence, ProSave and in turn the ProCom component model supports incremental construction. As with basic-pnf from Section 3.5.1, in a ProSave component, a triggered service reads data from in ports, computes and produces data on out ports. Unlike basic-pnf, one or more services in a component may be triggered independently; triggered services execute concurrently.

### 3.8.3 C2 Component Model

C2 (from Chiron-2), primarily proposed to support GUI based applications, is a publish-subscribe component model. In this style, concurrent components are composed by message routing devices (connectors). In C2, components (boxes) are connected to one or two connectors. Components/connectors have a top and a bottom interface (port). A component’s top interface can be connected to a connector’s bottom interface and a component’s bottom interface can be connected to a connector’s top interface. A typical system construction in two steps is shown in Figure 3.28.

![Figure 3.28: System construction in C2](image)

In the system, notifications travel downward in the system hierarchy and requests travel in the opposite direction. A connector in C2 is a bus which broadcasts requests/notifications to its connected components. C2 supports incremental construction as no existing behaviour of the system is destroyed by adding new increments.

### 3.8.4 SOFA Component Model

SOFA 2.0 is a successor of the SOFA component model [86] which supports all three kinds of ADLs studied in Section 3.5.1. For client-server style, SOFA 2.0 provides a
procedure (or method) call connector. A component can invoke a method of another component by using a procedure call connector through the connected ports. Interactions through the ports are specified by traces of method calls and returns; these traces are referred to as behavioural protocol for the component. A message leaving out of a port is referred to as emit activity on the port and similarly a message coming into a port is referred to as accept activity on the port.

A composite of two components is shown in Figure 3.29(a). The behaviour protocol of two components is shown in Figure 3.29(b). Unlike basic-cs from Section 3.5.1, in SOFA 2.0, a provided port may be connected to more than one required port. Incremental construction is supported in SOFA as adding increments does not destroy a system’s existing behaviour.

![composite components diagram](image)

Figure 3.29: System construction in SOFA 2.0

### 3.8.5 Pluggable Software Units

In [15], a component model has pluggable software units (PUs) that provides a realisation of independent and reusable software. In this component model, there are two types of component: basic and network. Components are composed hierarchically and composite components are indistinguishable from atomic components. Reuse of a component is achieved by removing any external dependencies from the component definition; this is achieved by making actual invocations of external actions via the output ports. Components have strongly typed gates with input/output signatures. Components with matching gates are composed by port connection and components with non-matching gates can be linked by an adapting filter.
3.8. A REVIEW OF RELATED WORK

A composite component is shown in Figure 3.30. Constructing a network component incrementally does not destroy the behaviour of any component; hence, this component model supports incremental construction. The interaction between two components corresponds to the client-server style, as defined in Section 3.5.1. However, unlike basic-cs, a PU component may have none, one or more components connected on its out ports; the default function of an out port returns nil if no PUs are linked to the port.

3.8.6 Reo Connectors

Reo [7] is a channel-based coordination model for the composition of concurrent processes (components). In this language, active components (not defined in Reo) communicate via channels (streams) for data exchange. A component is a non-empty set of active entities. The purpose of Reo is to coordinate data between components; active entities in a component can perform data input/output operations through channels connected to the component. Reo channels are different in their specifications which impose different coordination patterns for connectors.

In Figure 3.31, a system is constructed in two steps by using Reo Sync channels [8]. Components A and B produce data and component C consumes data. Constructing a system does not destroy the behaviour of any component; hence, Reo supports incremental construction.

In Reo, primitive channels can be composed to create a connector, as shown in Figure 3.31. A data channel-based Reo connector [7] imposes a specific coordination pattern on components to perform I/O operations.
3.9 Chapter Summary

In this chapter, in the context of CBD, incremental construction is defined to construct component-based systems. Next, we have studied and evaluated current component models in which components are objects, architectural units or encapsulated components. Based on the fixed behaviour of components, the X-MAN component model is selected for achieving incremental construction in CBD.

In the next chapter, we will describe the X-MAN component model in detail, which will help us to propose extensions in the model and the construction guidelines to achieve incremental construction in the extended X-MAN component model.
Chapter 4

The X-MAN Component Model

In this chapter, first the conventional X-MAN component model is described briefly which is necessary to understand the proposed extensions (in Section 4.2) to X-MAN. By ‘conventional X-MAN’ we mean the version of XMAN as presented in [67, 71]. Next, as part of the contribution of this thesis, we propose a number of extensions to the conventional X-MAN component model to further support incremental construction of practical systems. The extended X-MAN is referred to as X-MAN′. For X-MAN′ components, our proposed extensions include distinguishing components for user interactions from the components with computational behaviour, introducing annotations for services in a component’s interface, and the constrained output parameters for a service. For exogenous connectors in X-MAN′, our proposed extensions include the concept of service propagation and behaviour containment, open composition connectors and the flow constraint language.

In X-MAN, exogenous connectors play a central role in a system construction. In the conventional X-MAN, the semantics of exogenous connectors defined in [66, 82, 100] only address the control flow of composition connectors. In this thesis, we define the static and dynamic semantics of all exogenous connectors with our proposed extensions. In order to define the static semantics of exogenous connectors with the proposed extensions rigorously, we define interface generation (in the deployment phase) by exogenous connectors schematically. Moreover, for dynamic semantics (control and data flows in the run-time phase) of exogenous connectors, we use Coloured Petri net (CP-net) [53].

In this chapter, in order to illustrate the semantics of each connector individually,
trivial examples are used. However, to illustrate the meaningful use of these connectors, a simple ATM system is also considered. Exogenous connectors with the described behaviour are implemented in our exogenous composition framework (ECF).

4.1 The Conventional Model

The conventional X-MAN component model is a general purpose component model which can be used to construct component-based sequential systems. The model defines the atomic component and exogenous connectors to compose components. Moreover, X-MAN defines adaptor connectors to adapt components. In an X-MAN system, components and connectors are passive elements which perform their behaviours on receiving the system control initiated for a service request by the system user. For example, in the bank system in Figure 2.11 (on page 39), connectors G1 and SEL1 as well as the two bank components are passive when the ATM component is in the process of authenticating a client’s card details.

In this section, the concepts of the conventional X-MAN components and connectors are briefly described which are necessary to understand the proposed extensions (in Section 4.2) to X-MAN. The semantics of exogenous connectors (with the proposed extensions to X-MAN) for component interface generation and control/data flows will be defined/explained in Section 4.4 and in Section 4.5.

4.1.1 Atomic Components

Atomic components in X-MAN encapsulate computation and data [69]; encapsulating data means the data is private to the component, and encapsulating computation means execution of computation is only possible through the interface (provided port) of the component. Access to data is possible through the services of the component; this way an atomic component maintains its state. Moreover, X-MAN components do not support re-entrance, i.e., making another service request to a component while the previous request of a service is still processing [97].

An atomic component can be defined as a tuple \( \langle N, IU, U, D, PN \rangle \) of component name \( N \), invocation connector \( IU \) (an exogenous connector), computation unit \( U \), data \( D \) and provided interface of the component \( PN \), as shown in Figure 4.1(a).

In an atomic component, the computation unit is a set of methods or procedures and
4.1. THE CONVENTIONAL MODEL

A method can accept an arbitrary number of arguments and can return an arbitrary number of values (result of computation). A method can be defined as a pair \((mSig, \text{comp})\) of method signature \((mSig)\) and method computation/implementation \((\text{comp})\). The method signature can be defined as a 3-tuple \(\langle mN, oList, iList \rangle\) of method identifier \((mN)\), a list of output data types \((oList)\) and a list of input data types \((iList)\).

In an atomic component, the invocation connector (defined as a one-to-one correspondence function in Figure 4.1(b)) creates the provided interface of the component and invokes methods from the computation unit. The interface of a component is a set of service signatures where a service corresponds to one method in the computation unit; for a component, a service is a unit of computation. A service’s signature is a 3-tuple \(\langle sN, oList, iList \rangle\) of service identifier \((sN)\), a list of output data types \((oList)\) and a list of input data types \((iList)\).

The behaviour of an atomic component (Figure 4.1(c)) is the set of services exhibited by the component, as described in Section 3.1.2. A service request \((\text{Req})\) is a pair \(\langle sN, iList \rangle\) of the service identifier \(sN\) and a list of arguments \((iList)\) for the service. Similarly, a service response \((\text{Res})\) is a pair \(\langle sN, oList \rangle\) of the service identifier \(sN\) and a list of output values \((oList)\).

In an atomic component, for a service in the component’s interface, the invocation connector keeps a record for the service association with exactly one method in the connected computation unit. During run-time, for a service request to an atomic component, the invocation connector of the component accepts the service request and
initiates a method call (by passing arguments from the service request) for the associated method in the computation unit. A method in a computation unit can only call methods from the same computation unit. After receiving the results of a method execution, the invocation connector of the component creates a response for the requested service. In a component, the actual computation lies with methods in the computation unit. Hence, a service request of a component is mapped to the associated method call.

4.1.2 Exogenous Connectors

In X-MAN, for system construction, exogenous connectors are defined [102] to support control structures for sequencing, branching and looping. Moreover, unary connectors are defined to adapt components. These different kinds of exogenous connectors are referred to as types of exogenous connectors. In Figure 4.2, exogenous connectors are categorised into two groups: (i) composition connectors (e.g. sequencer, pipe and selector) and (ii) unary connectors (e.g. guard and loop). Unary exogenous connectors are called adaptors.

(a) Composition connectors

(b) Adaptors

Figure 4.2: Exogenous connectors

Unary Connectors

 Unary or adaptor connectors are used in system construction. An adapted component is a pair \( \langle UCon, C \rangle \) of a unary connector \( UCon \) and a connected component \( C \). A service request \( (Req) \) to the connected component and the service response \( (Res) \) from the component are routed by the adapting connector.

A guard connector corresponds to the selection construct to pick a single action in an imperative language, e.g. ‘if (condition) then do-action;’. If the guard condition is failed, then the connector does not forward a service request.

The loop connector corresponds to the repetition construct to repeat an action in an imperative language, e.g. ‘for (i=0 to 10) do-action;’. In X-MAN, there are two
loop connectors *infinite loop connector* and *finite loop connector*. Adapted by a finite loop connector, an adapted component can further be composed; otherwise the adapted component cannot be composed further. An infinite loop connector can only be used as the root connector in a system. For example, in the bank system in Figure 2.11 (on page 39), infinite loop connector \(L_1\) makes the system ready to accept another request after serving a request.

In an adapted component, for a service in the component’s interface, the adaptor connector keeps a record for the service association with exactly one service in the connected component. During run-time, for a service request to the adapted component, the adaptor connector of the component accepts the service request and initiates a service request (by passing arguments from the service request) for the associated service in the connected component. After receiving the results of a service request, the adaptor connector of the adapted component creates a response for the original requested service. In an adapted component, the actual computation lies with services of the connected component. Hence, a service request of an adapted component is mapped to the associated service of the connected component.

**Composition Connectors**

Exogenous composition connectors [71] are n-ary connectors that coordinate the flow of control and data to the composed components. There are three basic generic composition connectors in X-MAN: (a) sequencer, (b) pipe and (c) selector, as shown in Figure 4.2. A composite component can be represented as a tuple \(\langle CCon, Com \rangle\) of an exogenous composition connector \(CCon\) and a list \(Com\) of two or more composed components. In order to depict the working of composition connectors, we consider two atomic components with one service each; during composition, the composing connector creates the interface for the composite component. The creation of the interface for the composite component by the three composition connectors is shown in Figure 4.3.

A sequencer connector composing two components \(A\) and \(B\) (with \(p\) and \(q\) services, respectively) creates a composite with fixed \((p \times q)\) number of compound (or combined) services, as shown in Figure 4.3(a)(i). In a sequencer composite, a request for a compound service invokes one service from each composed component. For the requested compound service, the sequencer connector (composing two components \(A\) and \(B\) in Figure 4.3(a)(ii)) coordinates five activities in order: (i) creates a service request for component \(A\), (ii) gets the results of execution from \(A\), (iii) creates a service
CHAPTER 4. THE X-MAN COMPONENT MODEL

request for component B, (iv) gets the results of execution from B, and (v) creates a response (from the results of executions from components A and B) for the compound service request.

As with the sequencer, a pipe connector composing A and B creates a composite with fixed \( (r = p \times q) \) number of compound services (Figure 4.3(b)(i)). For a compound service request in a pipe composite, the connector invokes one service from each composed component. However, unlike the sequencer, results of execution generated by a service of first component can be passed as input data in the service request to the second component. For example, two components \( C_1 \) (with only one service \( x \) to return one output of type \( \text{int} \)) and \( C_2 \) (with only one service \( y \) to accept \( m \) inputs of type \( \text{int} \)) are composed by a pipe to create a composite \( C_3 \). For the compound service in \( C_3 \), the output value of sub-service \( x \) can be passed as argument to one or more input parameters of sub-service \( y \); this is referred to as data flow between two composed components. Based on data flow between two components of composite \( C_3 \), there are many different possible instances of \( C_3 \) in a system. For a composite shown in Figure 4.3(b)(ii), there is only one possible instance of the composite. For the requested compound service, the pipe connector (composing two components A and B in Figure 4.3(b)(ii)) coordinates five activities in order: (i) creates a service request for component A, (ii) gets the results of execution from A, (iii) creates a service request
(having some values from the service results of previous component) for component \( B \), (iv) gets the results of execution from \( B \), and (v) creates a response (from the results of executions from components \( A \) and \( B \)) for the compound service request.

Unlike the sequencer and pipe connectors, a selector connector composing two components (e.g. \( A \) and \( B \)) creates a composite with different number of services. A service request to such a composite invokes one service from only one composed component. Similar services of composed components (e.g. withdraw service of two bank components in Figure 2.11 on page 39) are referred to as matched services. For the composite’s interface, the selector connector includes all services of the first component and includes the unmatched services of the other composed components. Based on a condition, the selector decides to route control (and data) to only one of the many composed components (Figure 4.3(c)(ii)). After execution of a service, the execution results are returned by the selector.

In a composite component, for a service in the component’s interface, the composition connector keeps a record for the service association with one or more services of the composed components. During run-time, for a service request to the composite component, the composition connector of the component accepts the service request and initiates one or more service requests (by passing arguments from the service request) for the associated service(s) in the connected components. After receiving the results of the service request(s), the composition connector of the composite component creates a response for the original requested service. In a composite component, the actual computation lies with the services of the connected components. Hence, a service request of a composite component is mapped to the associated service(s) of the connected component(s).

### 4.1.3 Composition in X-MAN

The idealised component life cycle [60] addresses the life cycle of components in three phases: (i) design phase, (ii) deployment phase and (iii) run-time phase. In the design phase, atomic components are coded and composite components are created by composing existing atomic/composite components from a component repository. Components created (coded/composed) in the design phase are system independent; these components are stored in a component repository which can be used to create different systems. By system independent we mean a component which can be used for constructing different systems that require the component’s computational functionality. In the deployment phase, new system independent components cannot be
created for the repository but systems are constructed by using existing components and connectors from the (component/connector) repository. A composite created in the deployment phase is no longer system independent. For example, in a selector composite, a selection criterion added to the selector connector makes the composite system specific. For the simulation of such X-MAN composites, different tools (used in [61, 66, 102]) make the composites system specific at the implementation level differently in unspecified ways. In the run-time phase, to execute a system in a run-time environment, the system components/connectors are instantiated and control/data flows between the system components.

In the context of the idealised component life cycle, X-MAN supports component composition in the design and deployment phases as shown in Figure 4.4; however, composition in these two phases has different semantics. Unlike design phase composite components, a composite component in the deployment phase has no enclosing box; this means the composite is open for further extension or composition. A repository of pre-developed components and connectors plays a central role in the design and deployment phases.

![Figure 4.4: Composition in X-MAN](image)

In the design phase, a builder tool is used to create generic (system independent) composite components for further composition. An assembler tool is used to create a composite/system for a particular run-time environment (RTE) in the deployment phase. Composites created during both phases are self-similar with respect to the composite’s architecture; these composites have a root exogenous connector to coordinate
the control/data flow to the composed components.

### 4.1.4 System Construction

Constructed in the deployment phase, the architecture of a system is a collection of components and connectors, as shown in Figure 4.5. Structurally, a system grows horizontally if a component is added and the system grows vertically if a connector is added. By adding a composite, the system grows in both directions.

A system architecture can be separated into two layers for computation and control. Exogenous connectors in the control layer can be separated into many levels, as shown in Figure 4.5. This is possible because of the hierarchical nature of composition in X-MAN. In a system, in the final layer, there is only one root exogenous connector.

In this thesis, for incremental system construction, we focus on components, connectors and composites in the deployment phase. For system construction in the deployment phase, system developers view the system as a structure of components, adapted components/composites and composites as shown in Figure 4.6. Exogenous connectors treat components, adapted components/composites and composites alike; hence, we refer to (adapted) components or composites as components.

Components in the conventional X-MAN are free of re-entrance problem [97], as there is only one system execution thread. The root infinite loop connector (as shown in Figure 4.7) enables the system to entertain the next client after serving the previous
4.1.5 The Bank Example

In order to illustrate the role of different exogenous connectors (composition connectors and adaptors), we reconsider the bank example from Section 2.5.3. The bank system (Figure 4.7) has one ATM (Automated Teller Machine) to serve two banks. To create a bank composite, two bank components are composed using a selector exogenous connector SEL1. SEL1 routes the request to only one of the composed components at a time. Then the composite of two bank components is adapted with a guard connector G1. The adapted composite is then composed with the ATM component by using the pipe connector PIPE1. In the end, the composite is adapted by an infinite loop connector L1.

During the system execution, the ATM component reads an ATM card and pin code to authorise a customer. PIPE1 passes the system control, bank details and result of
4.2. The Extended X-MAN Component Model

In this section, for incremental system construction, we propose to extend the X-MAN component model in a number of ways. First, we identify and describe a number of limitations of the conventional X-MAN; next, without changing the basic semantics, we propose extensions to X-MAN components and connectors. The motivation behind these extensions is to achieve further support for incremental construction.

4.2.1 Limitations of X-MAN

In this section, we identify the limitations of X-MAN that we address in our proposed extensions to the model.

The conventional X-MAN does not distinguish between purely computational components and components that interact with the external environment (to communicate data from/to the input/output devices). Being an abstract component model, X-MAN does not define the service matching mechanism for selectors. For the simulation of X-MAN components/systems, different tools (used in [61, 66, 102]) deal with the services matching criteria for a selector at the implementation level differently in unspecified ways. To use a service output of a guard adapted component from a repository, there is no indication for the developer to know whether the required service was actually executed or not. We address these three issues by proposing extensions to X-MAN components as described in Section 4.2.2.

In [38], Elizondo and Ndjatchi discussed the issue of meaningless services of a composite. For a composite component created by the sequencer and pipe composition connectors, all possible combinations of services from the composed components are included as the services for the composite. Hence, during system construction, services of the system and the sub-composites can grow in number rapidly and meaninglessly; this issue is raised for a basic composite in [38] as in the composite’s interface only 4 out of 64 services make sense. It is the responsibility of the system developer to select valid services from this set of services. We address this issue by introducing the property of selected services in exogenous connectors as described in Section 4.2.3.
In the conventional X-MAN, once a connector is added to a system with a fixed arity, the connector does not allow us to compose any more components. To address this issue, we proposed the concept of open composition connectors as described in Section 4.2.3.

In some component models (e.g., UML2.0, Palladio component model [20], and MechatronicUML [19]), state charts or state machine diagrams are used to represent the behavioural model of a component/system. In contrast, X-MAN provides a single system model that represents the architecture of the system; this system model is structure centric. The exact behaviour of exogenous connectors, e.g., the pipe connector for passing data between composed components and the selector connector for selecting a component, is not part of the system model. For example, in Figure 4.7, the selection criterion of the selector connector SEL1 to connect to a bank component and the data flow (from one service of ATM to a service of bank component) by the pipe connector PIPE1 are not defined in the model. For the simulation of X-MAN components/systems, different tools (used in [61, 66, 102]) addressed these issues at the implementation level differently in unspecified ways. In order to address this issue, we propose flow constraint language (in Section 4.2.3) to write constraints for such exogenous connectors. Moreover, in the system model, we annotate the connectors with their constraints.

4.2.2 Extending X-MAN Components

In this section, for system construction in the deployment phase, a different kind of atomic component (with computation to communicate data with the external environment) is proposed; this component is referred to as a user interface (UI) component. Moreover, we propose that methods and input/output parameters for methods in the computation unit are annotated with extra information. These annotations are helpful to define the composition mechanism of the selector connector explicitly and can be useful when searching for components in a repository of components. Furthermore, output parameters of a service can be either constrained or unconstrained; such an extension is needed for the system designers when they are using the services of a guard adapted component.

In this section, before describing the details and rationale for the aforementioned extensions, we present a conceptual model of X-MAN components with our proposed extensions.
4.2. THE EXTENDED X-MAN COMPONENT MODEL

Conceptual Model of X-MAN Components

Using UML class diagram notation, we create the conceptual model for X-MAN components as shown in Figure 4.8. The purpose of this conceptual model is not to represent all features/details of X-MAN but to show few features necessary to depict our proposed extensions (shown in the dashed rectangles in Figure 4.8) for X-MAN components.

Atomic components are developed in the design phase; an atomic component is one of two types, computational components and UI components. Computational (and composite) components can be composed in the design phase to create composite components for further composition. In contrast, UI components cannot be composed in the design phase; rather, they are used in the deployment phase to create systems (composites).

An atomic component has one computation unit with one or more methods. A method can accept zero or more input parameters and return zero or more output parameters (to represent the typed values returned by the method). In a service signature, an output parameter can be either constrained or unconstrained. For services offered by guard adapted component, constrained (output) parameters may not have any computation result but a default value (e.g. ‘null’) if the service request is not forwarded by the guard. In order to support the service matching mechanism for the selector connector, the computation method in the computation unit and its parameters can have one or more annotations. These annotations can be helpful in searching the right component from the repository.

Figure 4.8: A conceptual model of X-MAN components
User Interface (UI) Components

In the conventional X-MAN, despite using components to interact with the hardware devices (e.g. drink vending machine in [60]), components with pure computations (e.g. components that simply consume arguments and return values though the provided interface) are not distinguished from components interacting with the hardware devices. In this section, we explain that the distinction between such components is important in the context of system construction in the deployment phase.

In X-MAN, the infinite loop connector cannot be used as a root level connector if the system has services (offered in the interface created by the root connector) that require input arguments for service execution. For example, suppose the composite of two bank components (from Figure 4.7) is adapted by an infinite loop connector, as shown in Figure 4.9(a). This construction is meaningless as the withdraw service of the composite requires input arguments. In this composite, initially a request can be made with a specific account code and amount to withdraw; after executing the service, the root connector reiterates the same service with the same arguments infinitely.

In contrast to the use of the infinite loop connector in Figure 4.9(a), the use of this connector in a system is purposeful (legitimate) if a service exhibited by the root connector does not have any input arguments. For instance, the use of the infinite loop in the Figure 4.9(b) (an extended version of the bank system from Figure 4.7) reads data from UI components CR (card reader), PR (pin reader) and RR (request reader) which are linked with the input devices. The component CB (central bank) authorises the entered card and pin code. After serving one customer, the root infinite

![Figure 4.9: Infinite loop in a system](image)
4.2. **THE EXTENDED X-MAN COMPONENT MODEL**

loop connector transfers the system control to component CR. Unlike the system in Figure 4.9(a), in the bank system in Figure 4.9(b), for each withdrawal from a bank component, different data is accepted from the UI components.

For human system interactions, UI components consist of functionality to read/write from/to the input/output devices. For example, UI component CR (in Figure 4.9(b)) reads an ATM card data inserted into an input device (hardware unit to accept an ATM card). As with a computational component, a UI component is a passive component with computation (to interact with hardware devices) which does not call computations of other components. Hence, the fundamental semantics of the conventional X-MAN component model is not changed/violated. However, distinguishing a UI component from a computational component helps to reason about the use of the infinite loop connector, as explained by its use into two systems shown in Figure 4.9.

**Signature Annotations**

In X-MAN, the selector connector composes two or more components to create a composite with atomic/compound services. As explained in Section 4.1.2, a compound service is created by matching services from the composed components. Being an abstract component model, X-MAN does not define the service matching mechanism for selectors. For the simulation of X-MAN components/systems, different tools (used in [61, 66, 102]) deal with the services matching criteria for a selector at the implementation level differently in unspecified ways.

A service’s signature is a 3-tuple \( \langle sN, oList, iList \rangle \) of service identifier \( sN \), a list of output data types \( oList \) and a list of input data types \( iList \). Two components may have the same services with different service identifiers/names; for the composition of such components by a selector connector, matching the service identifiers (names) is not needed. Similarly, two components may be offering different computations with the same signatures. Then the question is how to match the services of such components by the selector connector? Since a selector connector may be used to compose at different connector levels in the connector hierarchy (as shown in Figure 4.5), therefore the service identifier itself may not be significant for the service matching mechanism.

To support the service matching based on some semantics (e.g. service descriptions, pre-/post-conditions etc.), annotations are proposed for methods and for the input/output parameters of methods in a computation unit. For example, to illustrate the concept of signature annotations, the bank component (from Section 4.1.5) is considered with two services, as shown in Figure 4.10.
Annotations are used to describe the purpose of the ‘withdraw’ service and its input/output parameters. In Figure 4.10(b), ‘mAnn’ represents a method annotation, ‘pAnn’ represents an input parameter annotation, and ‘rAnn’ represents an output parameter annotation. An annotation has properties where a property is a pair of name and string value; more annotations and more properties to an annotation can be added.

Signature annotations for services may be used for different purposes. For example, to help a developer in selecting a component from a large repository of components. For the selector connector, these annotations may be used in defining a mechanism for matching services from different components. For simplicity, we use annotations to write short descriptions for a service and its input/output parameters.

Constrained Output Parameters

In a service signature, an output parameter represents a member of returned result set generated by the execution of the associated service. Output parameters can be one of two types: constrained and unconstrained. Output parameters for services in an atomic component are always of unconstrained type, whereas output parameters for services of an adapted component by a guard connector are always of constrained type. In the deployment phase, a guard connector simply makes the value ‘null’ for each parameter of the result set of the requested service if control is not passed to the component (to represent that the parameter has no result).

In order to understand the reason why such a distinction is required, here we
consider the partial version of the bank system (from Figure 4.9(b)) shown in Figure 4.11(a). A parameter enclosed between brackets (‘[ ]’) is a constrained output parameter, as shown in the signature for the service of the adapted component G1. The value of a constrained output parameter can either be ‘null’ or a result generated by the service.

Based on the successful authentication of the customer’s bank card by CB, the partial system reads the amount to withdraw by executing a service of the UI component RR. In order to complete the system, the developer needs to add the bank component and to allow passing the amount read by RR to the bank component for cash withdrawal. First, assuming that all output parameters are of unconstrained type, a developer increments the system by adding component Bank1, as shown Figure 4.11(b). The system will crash if the customer’s authentication fails because the pipe connector has no result from RR to pass to Bank1.

Now, we consider two types of output parameters. Because of the presence of the guard, the developer knows that the output parameter of the guarded component is constrained by the guard’s condition (e.g., the use of a customer’s authentication from CB to connect to a bank component). Hence, the developer increments the system with a guarded component Bank1, as shown Figure 4.11(c). Guards G1 and G2 pass the system control to the component on successful authentication from CB. The system will not crash even if a customer’s authentication is failed. Thus distinguishing between constrained and unconstrained output parameters helps in constructing systems with
4.2.3 Extending Exogenous Connectors

We further extend X-MAN by augmenting the exogenous connectors from conventional X-MAN with a filter property to create an interface of selected services. Moreover, to allow us to increment composites, the composition connectors are extended to have open (increment-able) arity; with this provision, a composite can be incremented by adding more components. Furthermore, a prototype flow constraint language with minimal features is defined to fix the system behaviour to reason about the behaviour containment for incremental construction.

In this section, before presenting the details and rationale for the aforementioned extensions, a conceptual model of the exogenous connectors with our proposed extensions is presented.

Conceptual Model of Exogenous Connectors

Using UML class diagram notation, we create the conceptual model for exogenous connectors as shown in Figure 4.12. The purpose of this conceptual model is not to represent all features/details of exogenous connectors but to show few features necessary to depict our proposed extension to add constraints to some exogenous connectors. In this model, we also propose the pipe connector as an extended version of the sequencer connector. In this conceptual model, our proposed extensions are in the dashed rectangle in Figure 4.12.

Exogenous connectors in X-MAN are defined at an abstract level, e.g. a pipe connector denotes the notion of data passing between the components coordinated by the connector. However, the exact behaviours of these connectors are not fixed at the component model level. In the conceptual model of exogenous connectors (Figure 4.12), some of the exogenous connectors are defined to contain constraints; these constraints are used to define the exact behaviour of constrained connectors.

In X-MAN, connectors are either composition connectors or unary connectors. There are three composition (sequencer, pipe and selector) and three unary (guard, finite loop and infinite loop) connectors. The invocation connector is a unary connector that is used to define an atomic component. Invocation connectors are actually a part of the atomic component, whereas other connectors are the first class model elements. By first class model elements, we mean that such elements maintain their structural
existence (instantiated separately than atomic components) in the system during the deployment and run-time phases.

**Service Propagation and Behaviour Containment**

In X-MAN, the sequencer and pipe connectors always create a fixed number of services (as described in Section 4.1.2) by combining all services from the composed components. This way of combining all services creates a large number of invalid services in the composite [38]. Generation of invalid services can be avoided by selecting services for propagation from the lower level to the higher levels of the connector hierarchy (shown in Figure 4.5). By being able to filter out invalid services, developer effort in looking for valid services from a long list of unwanted/invalid services can be reduced. We propose to add a filter property to the set of exogenous connectors to filter the undesired services and only include desired services in the component’s interface. For an exogenous connector, this property is a list of valid services that can be the interface created by the connector.

In order to ensure behaviour containment (as defined in Section 3.1.3) for incremental construction, in the deployment phase, it should be verified (manually or automatically) that a selected service (of a component) by a connector would be contained in the behaviour generated by at least one connector at a higher connector level. For example, in the system in Figure 4.13, two atomic components are composed by a level0 sequencer connector. The system is incremented by composing the system with...
an atomic component C.

Figure 4.13: Service Propagation

In the deployment phase, we assume that all services of atomic components (A, B and C) are propagated if no service is selected for propagation. In order to ensure behaviour containment for incremental construction, the system designer (manually or with the help of a tool) would verify that the selected compound service \( s_5 \) of SEQ1 (at level0) is contained by one of the selected services of SEQ2 (at level1). Because of the filter property, the composite of SEQ2 has combined services with service \( s_5 \) from the composite of SEQ1. In the interface generated by SEQ2, combinations of other services from SEQ1 are not possible as these service are not selected to be propagated to the next level connectors.

Open Composition Connectors

In [63], open composition connectors (with two ways to compose components, shown as dots in Figure 4.14) were defined as an extension to the conventional X-MAN component model. The motivation for this extension was to achieve system construction incrementally in many iterations.

With respect to the sequencer and pipe connectors, the approach with open composition connectors (shown in Figure 4.14) was limited. It allowed incrementing systems by adding a component/composite at the beginning or at the end of the control flow defined by the connector, as shown in Figure 4.14. For example, consider the open sequencer connector in the bank system (from Figure 4.9(b)) as shown in Figure 4.15(a). The partial version of the bank system shown in Figure 4.15(a) accepts the bank card (by component CR) and the pin code (by component PR). We increment the system by adding a component to read a thumb expression after pin code entry, as shown in
Figure 4.14: Open composition connector

Figure 4.15(b). Now the partial version of the bank system accepts the bank card (by component CR), then the pin code (by component PR), and finally a thumb expression (by component TE).

Similarly, a component can be added to perform its behaviour before component CR. However, adding a component in between two components composed by the open composition connector proposed in [63] is not possible. For incremental construction, it is not realistic to assume that the set of system requirements will always be written in an order that requires components to be added at the beginning or at the end of the process flow. Hence, to be able to construct systems from a realistic set of requirements, an open composition connector (sequencer or pipe) has to be replaced or refactored if a requirement demands, adding a component in the middle of the control flow defined by the connector.

Instead of proposing a completely new connector, in this thesis we further extend the open composition connectors of [63] to allow adding a component in the middle of two components composed by the same open composition connector. Hence, there is
no need to explicitly show open location points visually (by dots). The extended open composition connector is shown in Figure 4.16(a). In order to illustrate the use of such a composition connector, the bank system example from Figure 4.9(b) is extended by considering SEL1 to be an extended composition connector. A new bank component Bank3 is added to SEL1, as shown in Figure 4.16(b).

![Figure 4.16: Extended open composition connector](image)

In the remaining of this thesis, all connectors should be assumed to take this open form, unless otherwise specified.

**Connector Constraints**

At the system architecture level, some exogenous connectors (e.g. pipe, selector, guard and finite loop connectors) in the conventional X-MAN systems do not have fixed behaviour with respect to control/data flows (to execute a specific service) to the connected component(s). An exogenous connector generates the functional interface (a set of services) from which the desired services are selected by a developer. However, selecting services still does not fix the behaviour of a connector in a system.

A pipe connector is simply a sequencer connector with an ability to pass service results of a component to the service requests to other components composed by the pipe connector. However, at the model level, there is no indication of this data flow between the composed components. For a service request of a selector connector, the indication for the selection of a specific service of one of the composed components is not part of a system architecture. Similarly, the condition for forwarding a service request by a guard connector and for the repetition of service requests by a finite loop
connector are not mentioned in a system architecture. X-MAN is an abstract component model that does not address aforementioned issues in details. For the simulation of X-MAN components/systems, different tools (used in [61, 66, 102]) addressed these issues at the implementation level differently in unspecified ways.

To further elaborate the need for constraints for some exogenous connectors to fix their behaviour, we consider three composites of a pipe connector as shown in Figure 4.17.

![Figure 4.17: Connector and composite behaviour](image)

In a composite, a pipe connector passes the result of a service from one component as an argument to a service of another component. The composite shown in Figure 4.17(a) has one compound service which may pass the output value of service $a$ of component $A$ to service $b$ of component $B$. Assuming that the output of the compound service contains all the outputs of its sub-services, there are three possible ways to fix the behaviour of $AB$: (i) pass the result as the first argument of $b$, (ii) pass the result as the second argument of $b$, or (iii) pass the result as both the arguments of $b$. As a service of a component can accept/return an arbitrary number of values, the number of possible ways to fix a composite’s behaviour with such services can be extremely high for the connector in Figure 4.17(b). A component can have a number of services; hence, considering a composite with many services (Figure 4.17(c)) increases the number of possible ways to fix the behaviour even further.

In order to fix the behaviour of constrained exogenous connectors (pipe, selector, guard and finite loop) in a composite at the model level and to annotate the connectors in the model, we realise the need for a prototype language. As the flow of control/data through the connector is based on the connector constraints, we refer to this language as flow constraint language (FCL). The purpose of FCL is only to provide limited
features that are required to fix the behaviour of constrained connectors in a system.

A constrained connector is a virtual computer that translates its specific FCL con-
straint to define its interface generation behaviour in the deployment phase and con-
trol/data flows in the execution phase. Constrained connectors have their own specific
syntax/semantics for their FCL constraints; hence, the syntax/semantics of FCL for the
constrained connectors are defined in the respective subsections in Section 4.5.

4.3 Presenting Exogenous Connectors

In a system, exogenous connectors play dual role and that is to create the functional
interface of a component in the deployment phase and to define the control/data flow
to the connected component(s) in the run-time phase. Open composition connectors
(shown in Figure 4.12) are n-ary connectors. In a system, an open composition connec-
tor $CC$ with ‘n’ composed components (Figure 4.18(a)) is equivalent to a sub-system
of ‘$n – 1$’ $CC$ connectors of binary arity (Figure 4.18(b)) composed hierarchically.
Hence, the interface generation and control/data flow mechanisms of a binary compo-
sition connector of a specific type can be used recursively to achieve the behaviours
of an n-ary composition connector of the same type. Hence, for simplicity, we define
the aforementioned semantics for binary (rather than n-ary) composition connectors in
this chapter.

![Figure 4.18: Open composition connectors](image)

In order to define interface generation and control/data flow by exogenous con-
nectors, we consider minimum necessary features and define a connector as a tuple
4.3. **PRESENTING EXOGENOUS CONNECTORS**

\((\langle L, Z, P \rangle)\) of a set of selected service identifiers \((L)\), a set of (FCL) constraints \((Z)\) and a set of provided services \((P)\) offered by the adapted/composite component. In the deployment phase, one or more service identifiers can be added/removed from \(L\) for any connector. Set \(L\) can be empty which means all services are propagated by the connector. However, in the deployment phase, one or more constraints can be added/removed from \(Z\) for constrained connectors (pipe, selector, guard and finite loop) only. Hence, the set \(Z\) is always empty for unconstrained connectors (invocation connector, sequencer and infinite loop connector). Amongst the constrained connectors, the set \(Z\) is always non-empty for guard and finite loop connectors; however, the set \(Z\) may be empty for pipe and selector connectors. The interface generation mechanism of a connector adds elements (provided services) to the set \(P\). In the deployment phase, a service can be renamed in set \(P\).

Connector Template net (a special kind of Coloured Petri net (CP-net) [53]) in [66, 82, 100] defines the control flow for composition connectors. In this chapter, for exogenous connectors with extended features, interface generation semantics are shown schematically and the control/data flow to the connected/composed components are defined by using Coloured Petri net (CP-net) [53] semantics. CP-net and CPN tools are introduced with a simple example in Appendix B.1. We will use simple trivial examples to demonstrate both behaviours of a connector.

Transitions in a CP-net are triggered to simulate the net. Hence, in the description for a connector’s CP-net, we will describe the working of the CP-net with respect to its transitions. Some of the transitions are common amongst CP-nets for different connectors; we will avoid to repeat such transitions in our description. In order to verify the correctness of our CP-net for the exogenous connector, we will describe the simulation (of our CP-nets in the CPN tools) in Appendix B.

In the next two sections, for simplicity, first we define and present the semantics of unconstrained exogenous connectors. Next, we define and present the semantics of constrained exogenous connectors. To ease the readability, we present/explain semantics of exogenous connectors in a sequence with respect to their associativity with each other. For example, the pipe connector is an extended version of the sequencer connector and the selector connector is an extended version of the guard connector. Hence, pipe is defined after sequencer; similarly, selector is defined after guard.
4.4 Unconstrained Exogenous Connectors

In this section, we will describe the semantics of each of the unconstrained exogenous connector in turn. We will start by describing the interface that each connector presents to the world, and how it is constructed from the interfaces of the connected components. By “interface” here, we mean the set of services that the connector offers for invocation in the deployment phase. Next, we will describe the dynamic semantics of each connector that defines the flow of control and data through the connector to the connected component(s) in the run-time phase.

4.4.1 Invocation Connector

The invocation connector is a unary connector that is used to define atomic components in the design phase. There is no direct use of an invocation connector in the deployment phase for system construction. However, for completion, the behaviour of this connector to create the provided interface of an atomic component and to transfer control/data to/from the computation unit is defined in this section.

Interface Generation

For the invocation connector, set $L$ is always empty that means all services created by the connector are propagated to be as provided services by the atomic component. An invocation connector is an unconstrained connector; hence, set $Z$ is also empty for this connector. The interface generation semantics of the invocation connector is defined as a (one-to-one correspondence) function in Figure 4.1(b). Hence, the set of connector’s computations (offered as services) is the same as the set of connected computation unit’s computations (offered as methods). In Figure 4.19 (a), the invocation connector is defined as a function with an input set (of public method signatures from the connected computation unit) to produce an output set (of provided service signatures for the atomic component).

A method signature from the input set is a 3-tuple $⟨ mName, oList, iList ⟩$ of method name or identifier, a list of output data types and a list of input data types. For each method from the input set, there is a provided service signature in the output set. A provided service signature $pSig$ from the output set is defined as a 3-tuple $⟨ sID, sSig, mRef ⟩$ of service identifier (a unique positive integer value), service signature and the reference to the associated method from the computation unit. Element $sSig$ from the provided service signature is a 3-tuple $⟨ sName, oList, iList ⟩$ of
4.4. UNCONSTRAINED EXOGENOUS CONNECTORS

Figure 4.19: Interface generation

A simple example (Figure 4.19 (b)) is used to illustrate the behaviour of an invocation connector in the design phase. In this example, a computation unit $U$ with two public methods (add and double) is connected with an invocation connector $IU$ to create an atomic component $A$. $IU$ creates the interface $PA$ of component $A$. Method signatures in $U$ and service signatures in $PA$ are shown with information readable by a human user in the design phase. Rest of the details in the signature (shown in Figure 4.19 (a)) are for internal use by the connector (for component/system design and for system execution).

Control and Data Flow

During execution, on receiving a request message for a service from outside the connector, the invocation connector ($IU$ in Figure 4.20) initiates a method call (for a method in the computation unit $U$) with data received in the request message. The invocation connector then generates a response for the received request containing the results of computation returned by the computation unit.

The control/data flow semantics of the invocation connector are shown with the help of a CP-net inside the connector in Figure 4.20. A service request (submitted to the place $Req$) is a tuple $⟨ID, aList⟩$ of service identifier ($ID$) and a list ($aList$) of service names, a list of output data types and a list of input data types.

A simple example (Figure 4.19 (b)) is used to illustrate the behaviour of an invocation connector in the design phase. In this example, a computation unit $U$ with two public methods (add and double) is connected with an invocation connector $IU$ to create an atomic component $A$. $IU$ creates the interface $PA$ of component $A$. Method signatures in $U$ and service signatures in $PA$ are shown with information readable by a human user in the design phase. Rest of the details in the signature (shown in Figure 4.19 (a)) are for internal use by the connector (for component/system design and for system execution).

Control and Data Flow

During execution, on receiving a request message for a service from outside the connector, the invocation connector ($IU$ in Figure 4.20) initiates a method call (for a method in the computation unit $U$) with data received in the request message. The invocation connector then generates a response for the received request containing the results of computation returned by the computation unit.

The control/data flow semantics of the invocation connector are shown with the help of a CP-net inside the connector in Figure 4.20. A service request (submitted to the place $Req$) is a tuple $⟨ID, aList⟩$ of service identifier ($ID$) and a list ($aList$) of service names, a list of output data types and a list of input data types.
input arguments for the requested service. The transition $T1$ is enabled and fired as the set of selected services $L$ (a list variable in the CPN tools for the connector's CP-net) for the invocation connector is empty. Place $cI$ (with the colour for $pSig$ shown in Figure 4.19 (a)) in the CP-net maintains a list of provided services (created by the connector in the design phase as shown in Figure 4.19) of the atomic component. Transition $T2$ is enabled and fired if the requested service is found in $cI$. Method reference ($mRef$) from $pSig$ (from $cI$) for the requested service (along with the list of arguments) is forwarded to place $CPo$ and the requested service $ID$ is forwarded to place $mem$. The connected computation unit receives a method call from $CPo$ and executes the method and returns the result set to place $CPi$. Then transition $T3$ is enabled and fired to send a tuple ($\langle ID, rList \rangle$) of the requested service identifier ($ID$) and a list ($rList$) of result values to place $Res$.

In Figure 4.20, a request for a service to add (shown in Figure 4.19 (b)) two integer values (2,3) is executed and the result value (5) of the requested service execution is returned by the connector. The description of the invocation connector’s (shown in the
atomic component $A$ shown in Figure 4.19 and Figure 4.20) simulation in the CPN tools is provided in Appendix B.2.

### 4.4.2 Infinite Loop Connector

In a system, the infinite loop (Iloop) connector is a unary connector which sends control to the connected component forever. Such a connector is used as a root connector when the system is complete. In this section, interface generation and control/data flow semantics of Iloop are defined.

**Interface Generation**

The Iloop connector adapts a component; the interface generation semantics of Iloop with default connector properties (with $L = \{\}$) is shown as a function in Figure 4.21 (a).

![Figure 4.21: Interface generation](image)

To Iloop, input is a set ($cSigs$) of provided services from the connected component. A service signature in this set is a tuple ($\langle sID, sSig \rangle$) of service identifier ($sID$) and service signature ($sSig$). Output of Iloop is a set ($pSigs$) of provided services; this represents the provided interface of the adapted component. A service in $pSigs$ is a 3-tuple of ($\langle sID, sSig, sRef \rangle$) service identifier ($sID$), service signature ($sSig$) and a reference ($sRef$) to a service of the connected component. Service signature
CHAPTER 4. THE X-MAN COMPONENT MODEL

$s肽 is described in Section 4.4.1. Iloop is a one-to-one correspondence function. Hence, size (no of elements) of both sets ($cSigs$ and $pSigs$) is equal. However, unlike the invocation connector, the service signature of the connector is different than the associated service signature from the connected component. In our extension to distinguish UI components from computational components in Section 4.2.2, we described that there would be no argument ($|aList|=0$) in a request to Iloop. As the connector repeats a request indefinitely to the connected component, there would be no output to the user either. Hence, in a service signature ($sSigmoid$) in $pSigs$, list of output and input parameters ($oList$ and $iList$) are always empty.

In the deployment phase, a connector’s property for propagated services can be changed. The default property (with $L = \{\}$) propagates all services of the connected component. A system designer can change this property by adding only desirable services to be propagated into the set of provided services of the adapted component.

A simple example (Figure 4.21 (b)) of a composite component $A$ is used to illustrate the interface generation of the Iloop connector $IL1$. A more practical example of Iloop will be shown in a bank system (in Section 4.6). In this example (Figure 4.21 (b)), component $A$ is a composite of two UI components (as described in Section 4.2.2) $C1, C3$ and a component $C2$ (with a computation service). In the composite, $PIPE1$ connector reads two integer values from a user (a service of component $C1$) and passes them to component $C2$ for computation and passes the result to component $C3$ for display. In this example, the focus is on the Iloop connector; hence, constraint of $PIPE1$ is not shown/discussed. Iloop adapted components’ services have no output/input parameters.

Control and Data Flow

Iloop connector is different than other connectors as there is no response for a request. The control/data flow semantics of an Iloop connector are shown with the help of a CP-net inside the connector in Figure 4.22.

During execution, a service request from the provided interface of an adapted component is accepted by the Iloop connector. On receiving a request message for a service from outside the connector, the Iloop connector initiates a service request to the connected component.

CP-net (comprised of transitions $T1$ and $T2$ along with associated places $Req, Temp, cI$ and $CPo1$) of the pipe connector is exactly the same as with the invocation connector. Unlike the invocation connector, there is no $mem$ place as there would
be no response out of the connector. In our extension to distinguish UI components from computational components in Section 4.2.2, we described that there would be no argument (|aList|=0) in a request to Iloop. Service reference (sRef) from pSig (from place cI) for the requested service (along with the list of arguments) is forwarded to place CPo. On return of a response to place CPi from the connected component, transition T3 sends the request again to place CPo.

In Figure 4.22, a request for ‘add’ service to component A’ (Figure 4.21 (b)) is made. The description of the simulation (in the CPN tools) for the infinite loop connector (shown in the adapted component A’ shown in Figure 4.21 and Figure 4.22) is provided in Appendix B.3.

### 4.4.3 Sequencer

In a system, a sequencer defines the provided interface of a composite. For a service request of a composite created by a sequencer, the second role of the connector is to initiate a sub-service request from each composed component and to produce a joint response by collecting responses from the sub-services of the composed components.
Interface Generation

A sequencer connector (SEQ) creates a composite by composing two or more components. The interface generation semantics of a binary sequencer connector with default connector properties (with \( L = \{ \} \)) is shown as a function in Figure 4.23.

For SEQ, a composed component is taken as an input of set \( cSigs \) of provided services and output is a set \( pSigs \) of provided services for the composite. A member in set \( cSigs \) is described in Section 4.4.2. The provided services (in \( pSigs \) shown in Figure 4.23(a)) of a SEQ composite are combined (or compound) services created by combining one service (referred to as a sub-service of the combined service) from each composed component in the connected sequence from left (lower index) to right (higher index). A service \( (pSig) \) of the composite is a 3-tuple of \( \langle sID, sSig, sList \rangle \) service identifier, service signature and a list \( (sList) \) of sub-services’ IDs of the composed components in component connecting order. Service signature \( sSig \) in \( pSigs \) is described in Section 4.4.1.

The number of services \( (n) \) in the composite is equal to the product of number of services in the composed components. A service in the composite is associated to exactly one service from each composed component in the order of component connections. The pattern of associations of the composite service with the sub-services is shown in Figure 4.23(b). In a composite service’s signature, the list of output data types \( (oList) \) is a concatenation of lists of output data types of the associated sub-services in the order of component connections. Similarly, in a service’s signature of
the composite, the list of input data types \((iList)\) is a concatenation of lists of input data types of the associated sub-services in the order of component connections. In Figure 4.23(c), the pattern of associations between the composite’s services and the composed components’ services is generalised. This generalisation shows a composite of two components with \(a\) and \(b\) number of services respectively.

In the deployment phase, a composition connector property for propagated services can be changed. The default property (with \(L = \{\}\)) propagates all sub-service combinations as provided services of the composite. A system designer can change this property by adding only desirable combined services to be propagated into the set of provided services of the composite. This is described in the following example.

Two simple examples (shown in Figure 4.24) are used to illustrate the behaviour of a sequencer connector in the deployment phase. In the first example (Figure 4.24(a)), two components (\(A\) and \(B\)) with one service each are composed by a sequencer connector (\(SEQ1\) with default properties) to create a composite (\(AB\)). In the composite, a service is a combination of two services from the composed components. Service signatures in a component’s provided interface are shown with information readable by a human user in the deployment phase. Rest of the details in the signature (shown in Figure 4.23) are for internal use by the connector (for system design/execution).

In the second example shown in Figure 4.24(b), three components (\(C\), \(D\) and \(E\)) with two services each are composed by a sequencer connector (\(SEQ2\)) to create a composite (\(CDE\)). After creating the composite, the default property (\(L = \{\}\)) of \(SEQ2\) is modified. Two service identifiers (shown as members of set \(L\) in Figure 4.24(b)) are added into the propagate-able services of the connector. Hence, all services of the provided interface of the connector with default properties which are non-members of set \(L\) are simply disappeared from the provided interface.
**CHAPTER 4. THE X-MAN COMPONENT MODEL**

**Control and Data Flow**

During execution, a service request from the provided interface of a composite is accepted by the composition connector. On receiving a request message for a service from outside the connector, the composition connector (SEQ1 in Figure 4.25 for composite AB shown in Figure 4.24(a)) initiates service requests (for associated sub-services from the composed components) with data received in the request message. The response of each sub-service is stored in the connector and returned as a single response after receiving the response of the last sub-service.

![Diagram of control and data flow](image_url)

Figure 4.25: Control and Data Flow of SEQ1

The control/data flow semantics of a sequencer connector are shown with the help of a CP-net inside the connector in Figure 4.25. CP-net (comprised of transitions T1 and T2 along with associated places Req, Temp1 (corresponds to place Temp in the invocation connector’s CP-net), cI and mem) of the pipe connector is exactly the same (with one variation to transition T2) as with the invocation connector. Transition T2 passes the list of sub-services (sList) for the requested service along with the list of
service arguments \( (aList) \) to place \( subSrvs \).

For each connected component to a sequencer connector, there is a place \( (cI1 \text{ for } A \text{ and } cI2 \text{ for } B) \) that maintains necessary details of sub-services from the connected components; a token in these places is a tuple \( ((sID, argCount)) \) of a sub-service ID \( (sID) \) and total number of arguments \( (argCount) \) for the sub-service. Place \( bgInd \) has a token which is used to hold a value (0 for the first service) from where the list of input arguments will be read for a sub-service.

Transition \( T3 \) performs three tasks: (i) gets required number of arguments (from \( cI1 \)) for the first sub-service request, (ii) changes token value in \( bgInd \) (for the next sub-service), and (iii) initiates a sub-service request by passing a request token to place \( CPo1 \). The response of the first sub-service is received by place \( CPi1 \). Next, like \( T3 \), transition \( T4 \) gets required number of arguments (from \( cI2 \)) for the next sub-service request and initiates the request (by getting arguments for the service from \( subSrvs \)) to place \( CPo2 \). The response of the second sub-service is received by place \( CPi2 \). Finally, transition \( T5 \) collects responses of sub-services of all composed components and creates a response token for the requested service.

In Figure 4.25, a request for a service \( \text{addMul} \) (shown in Figure 4.24 (a)) with four integer values \( (2,3,4,5) \) is made. The response token has the result of addition for the first two arguments and has the result of multiplication for the other two arguments. The description of the simulation (in the CPN tools) of the sequencer’s CP-net (shown in Figure 4.24(a) and Figure 4.25) is provided in Appendix B.4.

4.5 Constrained Exogenous Connectors

In Section 4.2.3, we discussed the issue of fixing behaviour of some exogenous connectors and proposed to add a property to exogenous connectors to hold the connectors’ constraints. Hence, we propose a prototype language \( (FCL) \) with less-than-ideal features to fix the behaviour of exogenous connectors in the deployment phase. With these intended simplifications in \( FCL \), we are able to gain meaningful results within the scope of achieving incremental construction by using X-MAN$^{'}$.

In this section, first we define the EBNF (extended Backus-Naur form) \[105\] and describe the operational semantics of \( FCL \) constraints for the constrained connectors. Next we define the interface generation (in the deployment phase) and control/data flow (in the run-time phase) semantics of constrained exogenous connectors (pipe, guard, selector and finite loop connector). Constrained connectors’ semantics are based on
their constraints written in FCL defined for this thesis. Hence, the operational semantics of FCL constraints are part of the control/data flow behaviour of their respective exogenous connectors. Structural operational semantics (SoS) [87] of FCL constraints are defined in Appendix A.

In order to avoid unnecessary repetition and to increase readability of this section, in Figure 4.26, common terms of EBNF for FCL are shown.

![Common terms for EBNF](image)

Type sets associated with FCL constraints are shown in Figure 4.26(a). Elements of set $P$ and $ID$ represent locations to hold a service parameter value and the index of a component/service respectively. Sets $T$, $R$, $I$, $S$ and $Num$ represents actual values. The legend of EBNF for FCL constraints is shown in Figure 4.26(b).

### 4.5.1 Pipe

Pipe is a special sequencer connector with an FCL constraint; in a composite, pipe allows output values of the first component’s service to be passed as input service arguments to the second component. A pipe connector without any FCL constraint is simply a sequencer. In a system, generation of the provided interface of a composite by a pipe connector is dependent upon the FCL constraints of the connector. Similarly, control/data flows between the composed components is also dependent upon the FCL constraints of the connector.

In this section, EBNF for pipe constraints are defined/explained. Next, for a pipe connector, semantics of generation of provided interface and flows of control/data are defined. Operational semantics of pipe constraints are part of the control/data flow semantics of the connector.
EBNF for Pipe Constraints

The EBNF grammar rules of an FCL constraint for a pipe connector are shown in Figure 4.27. With the help of this constraint, the behaviour of a pipe connector in a composite is fixed, as described in Section 4.2.3. For a compound service (represented by \( sN \)) generated by a pipe connector, an FCL constraint is a collection of one or more data mapping expressions (for the associated sub-services) between the source sub-service’s output data of one component to the recipient sub-service’s input data of another component. Two rules must be followed in these constraints: the component of the source service is on the left hand side (at lower index) to the component (at higher index) of the recipient service and data types of the output parameters (of the source service) are assignable to the data types of the corresponding input parameters (of the recipient service). In a dataFlowExp, terminal “param” on the left side of terminal “=” represents a parameter from the recipient service’s input list and “param” on the right side of terminal “=” represents a parameter from the source service’s output list. In a data mapping construct, component name (represented by \( cN \)) and service name (represented by \( sN \)) separated by ‘.’ (used before terminal “::” in the data mapping rule) is the context (or prefix) of \( \langle \text{param} \rangle \) tokens used on the left hand side of ‘=’ token in every data flow expression (\( \langle \text{dataFlowExp} \rangle \)). Similarly, all data flow expressions in a data mapping construct (within curly (\{\}) brackets) are within the context of the compound service (prefix \( \langle sN \rangle \) of the data mapping construct) provided by the connector.

![Figure 4.27: EBNF of FCL constraints for a Pipe](image-url)

As an example, we assume that SEQ1 (from Figure 4.24(a)) is a pipe connector (as shown in Figure 4.28(b)) without any FCL constraint. For the composite’s service \textit{addMul}, a constraint (shown as input to parser function in Figure 4.28(c)) can be added
to the connector. A parser function (Figure 4.28(a)) in the connector parses the pipe constraint to a tuple (representing the abstract syntax tree for the constraint) of composite service ID \(csID\) and one or more tuples of data mappings. A data mapping tuple has six elements (receiver component index \(rcInd\), receiver service ID \(rsID\), receiver input parameter index \(ipInd\), source component index \(scInd\), source service ID \(ssID\) and source output parameter index \(opInd\)). For the compound service \(addMul\) of the composite, the shown constraint maps output result of \(add\) service of component \(A\) as first argument to \(mul\) service of component \(B\). In a constraint, numeric letters in alphanumeric strings (e.g. \(param0\)) for “param” tokens represent the location of a parameter in the list of input arguments (used on the left hand side of “=” token) and in the list of output results (used on the right hand side of “=” token).

![Diagram of a pipe constraint](image)

Figure 4.28: Example of a pipe constraint

For a service in a pipe composite, an FCL constraint maps output parameter(s) of the first component’s service as input parameter(s) to the second component’s service. The execution of a service is part of the connector’s semantics; hence, the operational semantics of a pipe constraint in the connector is to write contents of one location (an output parameter of a source service of the first component) to another location (an input parameter for a receiver service of the second component). The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the pipe connector for control/data flows to the composed components.

The pipe connector shown in Figure 4.28(b) with a constraint shown in Figure 4.28(c)
is defined as a CP-net shown in Figure 4.31. Transitions T5-T7/T7′ in Figure 4.32 define the operational semantics of a pipe constraint.

**Interface Generation**

In essence, the interface generation semantics of a pipe connector is an extension of interface generation semantics of a sequencer connector (shown in Figure 4.23 and described in Section 4.4.3). The extension needed for the pipe connector is shown in Figure 4.29. The limit of two services (one from each composed component shown in Figure 4.29) is considered for simplification as described in Section 4.3. However, in X-MAN′, a composition connector (used in Chapter 5 and Chapter 6) is n-ary that can compose two or more components.

A function $F1$ (in the pipe connector) accepts two service signatures (from the composed components) and the set $Z'$ of parsed FCL constraints; $F1$ creates provided signature of a compound service. This function checks (by calling a function $\text{rmParam}$) for existing parsed constraints in which an input parameter for a service is being mapped to an output parameter of another service. If such a constraint exists for an input parameter, that parameter is not considered in the list of input parameters for the compound service. Hence, input parameter list of a sub-service may be shortened in the signature of a compound service. In order to illustrate the extension behaviour, we reconsider the example from Figure 4.28. Now we consider that the FCL constraint shown in Figure 4.30(b) as a constraint of PIPE1 connector shown in Figure 4.30(a). According to the semantics of function $F1$ shown in Figure 4.29, first input argument of the $\text{mul}$ sub-service will not be part of the compound service ($\text{addMul}$) signature.
Hence, for the compound service’s signature generation, the input parameter list of sub-service \textit{mul} is shortened to length 1 from 2 by function \textit{F1}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_30.png}
\caption{Interface generation by a pipe connector}
\end{figure}

\section*{Control and Data Flow}

Control and data flow semantics of the pipe connector is also an extension of the control and data flow semantics of the sequencer connector. On receiving a request message for a service from outside the connector, the composition connector (\textit{PIPE1} in Figure 4.30) initiates service requests to the associated sub-services from the composed components. The response of the requested service is returned after receiving the response of the last sub-service. The control/data flow semantics of a pipe connector are shown with the help of a CP-net inside the connector in Figure 4.31. Subpage\footnote{Hierarchy feature of CPN tools provide a way to structure a complex CP-net into subpages (subnets).} for Group 1 is shown in Figure 4.32. First, we highlight the similarities of pipe’s CP-net with the sequencer’s CP-net. Then the extension for the pipe connector will be discussed.

CP-net (comprised of transitions \textit{T1}, \textit{T2} and \textit{T3} along with associated places \textit{Req}, \textit{Temp1}, \textit{cl}, \textit{mem}, \textit{subSrvs}, \textit{bgInd}, \textit{cl1}, and \textit{CPo1}) of the pipe connector is exactly the same as with the sequencer connector. Transition \textit{T9} in the pipe’s CP-net corresponds to transition \textit{T5} in the sequencer’s CP-net. For the pipe connector, the extension needed to the sequencer’s CP-net is the net that defines the semantics of passing result values (based on the FCL constraint) of the first sub-service as input arguments to the execution of the second sub-service. The difference of the two nets starts after receiving the
response of the first service in place $CPi1$. In the pipe connector, execution of transitions T1-T3 is the same as with the sequencer connector; hence, we start describing the net with transition $T4$.

Place $const$ has parsed FCL constraint tuple (shown in Figure 4.30 (b)). Transition $T4$ copies the output of the first sub-service to places $outputs$ and $Temp5$. For the requested service, transition $T5$ sends all data mapping tuples (between input arguments of a service and output values of another service) from the parsed tuple (from place $const$) to place $sConst$. Transitions $T4$ and $T5$ are shown in Figure 4.33.

Transition $T6$ (shown in Figure 4.34) is enabled if the count value in place $argCount$ (holds initially 0 to represent the count of input arguments of a sub-service) is less than the count of the input arguments (read from place $cI2$) for the next sub-service (a service from the second composed component) of the requested service.

Creating the input argument list for a sub-service request from the results of the previously executed sub-services (based on FCL constraints) and from the argument list of the compound service request is a critical task. $T6$ is part of the pipe CP-net which is responsible for creating input argument list for a request to the next sub-service. $T6$ can add/modify values in three places ($bgInd$, $Temp3$ and $Temp4$) concurrently. Place
bgInd holds a value for a parameter location from where the list of input arguments for the compound service will be read.

\( T_6 \) increments the value \( k \) (read from \( bgInd \)) if it finds a constraint (from place \( sConst \)) for an argument (at location \( k1 + 1 \) where \( k1 \) is a value from place \( argCount \)) of the next sub-service (from place \( cI2 \)). \( T_6 \) passes a tuple of argument list \( (al) \) from the service request, current count of sub-service input argument \( (k1) \), current index \( (k) \) of input argument list (of the compound service request) and an integer value 1 (if a constraint for an argument at location \( k1 + 1 \) in the sub-service is found) or 0 (if a constraint for an argument at location \( k1 + 1 \) of the sub-service is not found) to place
4.5. CONSTRAINED EXOGENOUS CONNECTORS

Temp4. T6 passes a 3-tuple of parameter mapping between two services (from place sConst), current count of sub-service input argument (k1) and an integer value 0 (if a constraint for an argument at index k1+1 is found) or 1 (if a constraint for an argument at index k1 + 1 is not found) to place Temp3.

Transition T7 (shown in Figure 4.35) is fired if the last element n in a tuple from place Temp4 is 0. This value shows that no constraint exists for the input argument at location k1 of the next sub-service. Hence, for this input argument, a value at index k is taken from the argument list al of the compound service request and appended into the argument list al1 from argsC2. Place argsC2 maintains the argument list for the next sub-service. T7 also increments the value in place argCount.
Transition $T7'$ (shown in Figure 4.35) is fired if three conditions are fulfilled: (i) the last element $n$ in a tuple from place $Temp3$ is 0; this value shows that a constraint exists for an input argument at location $k1$ of the next sub-service, (ii) the source component index from the parsed constraint ($ac$) is matched with the component index from place $outputs$, and (iii) the source service ID from the parsed constraint ($ac$) is matched with the service ID from place $outputs$. $T7'$ appends the argument list ($al1$ from $argsC2$) with a result value (taken from $outputs$) at a location (given in the parsed constraint $ac$) for the next sub-service. $T7'$ also increments the value in place $argCount$.

Transition $T8$ (shown in Figure 4.36) is enabled if value in place $argCount$ is equal to the count of input arguments (from place $cI2$) for the next sub-service. Transition $T8$ sends a request (to place $CPo2$) for the next sub-service (from place $cI2$) with argument list from place $argsC2$. The response of the second sub-service is received by place $CPi2$. Finally, transition $T9$ (shown in Figure 4.36) collects responses of sub-services of all composed components and creates a response token for the requested service.

In Figure 4.31, a request for a service addMul (shown in Figure 4.30 (a)) with three integer values (2,3,5) is made. The response token has the result of addition for the first two arguments and has the result of multiplication for the result of addition and the third argument. The description of the simulation (in the CPN tools) of the pipe connector (shown in Figure 4.31) is provided in Appendix B.5.
4.5. CONSTRAINED EXOGENOUS CONNECTORS

4.5.2 Guard Connector

In a system, a guard connector passes a service request to a component if the guard condition in an FCL constraint for the service is satisfied. Hence, interface generation and control/data flow semantics of a guard are dependent upon the FCL constraints of the connector. Output parameters of a guarded component’s services are constrained, as described in Section 4.2.2. FCL constraints of a guard can also add extra parameters in a service’s signature.

In this section, EBNF for guard constraints are defined/explained. Next, for a guard connector, semantics of generation of provided interface and flows of control/data are defined. Operational semantics of guard constraints are part of the control/data flow semantics of the connector.

EBNF for Guard Constraints

The EBNF grammar rules of FCL constraints for the guard connector are shown in Figure 4.37.

![Figure 4.37: EBNF of FCL constraints for a guard](image)

With the help of this constraint, the behaviour of a guard connector in an adapted component is fixed, as described in Section 4.2.3. For a service generated by a guard connector, an FCL constraint defines a condition (based on service input arguments) which is evaluated to forward the service request to the adapted component or not. A condition in the guard constraint is comprised of one or more Boolean expressions.
separated by logical operators ‘add’ or ‘or’. In a Boolean expression, a ‘param’ token represents the input argument at an index (e.g. ‘param0’ is an argument at index 0) in the list of input arguments for a service offered by the guard adapted component.

A parser function (Figure 4.38(a)) in the guard connector parses a constraint to a tuple (representing the abstract syntax tree for the constraint) of composite service ID \((sID)\) and a list \((conds)\) of conditions (or Boolean expressions) to forward the control/data to the connected component. A condition is a 5-tuple (shown in Figure 4.38(a)). Element \(lOp\) represents a logical operator between two conditions; for the first condition, \(lOp\) has value “null”. Element \(pInd\) represents an input parameter index of a service and \(pTy\) represents the data type of the input parameter at index \(pInd\). Element \(eOp\) represents the expression operator in a guard condition and \(val\) represents the actual value in the condition. Tuples in list \(conds\) are ordered by the value of element \(pInd\). In a condition, the data type of a service parameter \((pTy)\) is equal to the data type of the value \(val\) (represented by \(val.type\) in Figure 4.38(a)) in the condition. In a condition, the data type \((pTy)\) of an extra parameter (e.g. two bool parameters in ‘sal’ signature in Figure 4.38(b)) is inferred from the data type of the value \(val\) from the condition.

In order to illustrate the use of a guard constraint in a practical system, a simple adapted component (shown in Figure 4.38(b)) \(A\) with a service \(sal\) (to calculate the salary) is considered. The output of this service is a constrained parameter as described in Section 4.2.2. The first input argument is the number of worked hours and the second argument is the pay rate per hour. Guard \(G1\) forwards the control/data to the adapted component based on some values (of last two input parameters) not part of input arguments of the service of component \(A\); these arguments are passed by the
system. Hence, for service sal of component $A'$, a constraint (with extra parameters shown as input to parser function in Figure 4.38(c)) is added to the connector. A parser function in the connector parses this constraint.

For a service in a guard adapted component, the execution of a service is part of the connector’s semantics. Hence, an FCL constraint’s operational semantics is to evaluate the condition in the constraint. The evaluation of a condition with one or more Boolean expressions is a value from set $T$. The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the guard connector for control/data flows to the connected component.

The guard connector shown in Figure 4.38(b) with a constraint shown in Figure 4.38(c) is defined as a CP-net shown in Figure 4.40. Transitions $T2$-$T6/T6'$ in Figure 4.41 define the operational semantics of a guard constraint.

### Interface Generation

A guard connector adapts a component; the interface generation semantics of a guard connector with default connector properties (with $L = \emptyset$ and $Z = \emptyset$) is shown as a function in Figure 4.39.

- $cSigs = \{cSig_1, cSig_2, ..., cSig_k\}$
- $pSigs = \{pSig_1, pSig_2, ..., pSig_k\}$

**Where:**
- $cSig = \{sID, sSig, sID = i, sSig = sName, oList, iList\}$
- $pSig = \{pSig_1, pSig_2, ..., pSig_k\}$
- $sSig = \{sName, oList, iList\}$
- $oList$ = list of constrained output types
- $i_j \in \{\text{positive integers}\}$

**Note:** By default, $L = \emptyset$ and $Z = \emptyset$ for the Guard.

**Figure 4.39: Interface generation**

For the guard connector, a connected component is taken as an input of set $cSigs$ of provided services and output is a set $pSigs$ of provided services for the adapted component. Members in sets $cSigs$ and $pSigs$ are described in Section 4.4.2. In a guard adapted component, every service in the provided interface must have a constraint. Length of both sets $cSigs$ and $pSigs$ is equal. The guard connector is a (one-to-one correspondence) function between sets $cSigs$ and $pSigs$.

In the deployment phase, for a guard connector, the default property (with $L = \emptyset$) propagates all services of the connected component. A system designer can change this property by adding only desirable services to be propagated into the set of provided services of the adapted component.
In an adapted component by a guard connector, a service’s constraint is parsed by the connector as shown in Figure 4.38. For the provided services set $pSigs$ of an adapted component, the interface generation mechanism of the guard connector extends the $iList$ element of a service signature if there are extra parameters (not from the original service signature from the connected component) used in the service constraint (as shown in 4.38(b) for the constraint shown in 4.38(c)). This is the only possible involvement of a constraint in the interface generation mechanism of the guard connector. In the guard adapted component, for a provided service, the list of input parameters is the union of input arguments (of the associated service of the connected component) and extra parameters (used in the provided service constraint). The detailed process for the service signature extension with extra parameters is given in Appendix C.1.

**Control and Data Flow**

The control and data flow of a guard connector is shown in Figure 4.40.

During execution, on receiving a request message for a service from outside the connector, the connector ($G1$ in Figure 4.40) initiates exactly one service request (to
4.5. CONSTRAINED EXOGENOUS CONNECTORS

the connected component) with data received in the request message. The control/data flow semantics of a guard connector are shown with the help of a CP-net inside the connector in Figure 4.40; subpage for Group 1 is shown in Figure 4.41.

Transition $T1$ along with associated places $Req$, $Temp$ and $mem$ of the guard connector is exactly the same (with one variation to place the list of service arguments $aList$ to place $args$) as with the invocation connector.

![Figure 4.41: Group 1 of G1](image)

Place $Const$ has the parsed constraint for a service offered by the connector. Next enabled transition $T2$ gets the count of guard (Boolean) expressions from the requested service’s constraint from place $Const$ and writes that number to place $cCount$. $T2$ also writes the guard expressions from the guard’s condition to place $sConst$. Transition $T3$ passes a pair of the first guard expression and the index for the next expression to place $aCond$. Transition $T3$ is fired for each expression with a Boolean parameter in an expression from place $sConst$. Transitions $T2$ and $T3$ are shown in Figure 4.42.

The first element of a condition can be ‘null’ (processed by transition $T4$), ‘and’ (processed by transition $T4’$) or ‘or’ (processed by transition $T4”$). Transitions $T4$ is shown in Figure 4.43.
\( T_4 \) gets input argument list of the requested service from place \( args \) and checks the values of into parameters used in the service constraint from place \( aCond \) to evaluate a Boolean expression in the constraint. If the result of evaluation is successful then this transition writes a pair of numbers with first value 1 to place \( Temp1 \); otherwise, this transition writes a pair of numbers with first value 0 to place \( Temp1 \). As \( T_4 \) processes the first Boolean expression is the constraint, this transition disregards the evaluation of previous Boolean expression from place \( result \) (by default it has value 0 to represent \( false \)).

Transitions \( T_4' \) and \( T_4'' \) are shown in Figure 4.44. The function of these two transitions is exactly the same as with transition \( T_4 \) with one difference and that is to regard the evaluation of previous Boolean expression from place \( result \).
4.5. CONSTRAINED EXOGENOUS CONNECTORS

Figure 4.44: Transitions $T_4'$ and $T_4''$ in Guard CP-net

For $T_4'$, the (true) value from the result is joined with the processing (described for $T_4$) with a logical and operator. For $T_4''$, the (true) value from the result is joined with the processing (described for $T_4$) with a logical or operator.

Three transitions $T_5$, $T_6$ and $T_6'$ are shown in Figure 4.45. Transition $T_5$ sends the intermediate result (true/false) for processing an expression (by transition $T_4$ or $T_4'$ or $T_4''$) to place results. This transition passes the index for getting the next expression to place $nxtCond$.

Figure 4.45: Transitions $T_5$, $T_6$ and $T_6'$ in Guard CP-net
All expressions in place $sConst$ are processed in the explained way. If the final result of the service selection is false, then transition $T6'$ fires to write (an empty response) to place $Res$; otherwise, transition $T6$ fires to write to place $Temp2$.

Transition $T7$ fires the service request (for a service from place $CI1$ for the connected component) to place $CPo$. The response of this service is received in place $CPi$ from the connected component. Finally the transition $T8$ creates the response for the requested service to be sent out to the service requester. Transitions $T7$ and $T8$ are shown in Figure 4.46.

![Figure 4.46: Transitions $T7$ and $T8$ in Guard CP-net](image)

The description of the simulation (in the CPN tools) of the guard connector (shown in Figure 4.40) is provided in Appendix B.6.

### 4.5.3 Selector

The selector connector is an extension of the guard connector. However, unlike the guard connector, not all services (from the provided interface of a selector composite) have FCL constraints. For a selector composite, only compound services (services offered by more than one composed component) must have FCL constraints in the connector. A provided service request of a selector composite executes one service of a composed component. The service matching mechanism used by the selector connector is referred to as full-signature matching. In a system, generation of the provided interface and the control/data flows to the composed components of the selector composite may dependent upon the FCL constraint of the connector.

In this section, EBNF for selector constraints are defined/explained. Next, for a
selector connector, semantics of generation of provided interface and flows of control/data are defined. Operational semantics of selector constraints are part of the control/data flow semantics of the connector.

**EBNF for Selector Constraints**

The EBNF grammar rules for FCL constraints of a selector connector are shown in Figure 4.47. For the selector and guard connectors, the EBNF rules for the non-terminal \( \langle \text{condition} \rangle \) (shown in Figure 4.37) are same; hence, grammar rules for \( \langle \text{condition} \rangle \) are not shown in Figure 4.47.

![Figure 4.47: EBNF of FCL constraint for a selector](image)

With the help of a selector constraint, the behaviour of a selector connector in a composite is fixed, as described in Section 4.2.3. For a compound service generated by a selector connector, an FCL constraint defines the criteria (based on service input arguments) which is evaluated for the selection of a component for a service execution. The criteria for the selection of a component is comprised of a condition with one or more Boolean expressions separated by logical operators ‘add’ or ‘or’. In a Boolean expression, a ‘param’ token represents the input argument at an index (e.g. ‘param0’ is an argument at index 0) in the list of input arguments for a service offered by the selector composite.

A parser function (Figure 4.48(a)) in the selector connector parses a constraint to a tuple (representing the abstract syntax tree for the constraint) of composite service ID \( (sID) \) and the selection criteria for two components (represented by \( cSel_1 \) and \( cSel_2 \)). The selection criterion for a component is a tuple of component index \( (cInd) \) and a list \( (conds) \) of conditions (or Boolean expressions) to forward the control/data to the connected component. A condition is a 5-tuple (shown in Figure 4.48(a)). Element \( lOp \) represents a logical operator between two conditions; for the first condition, \( lOp \) has value “null”. Element \( pInd \) represents an input parameter index of a service and
\( pTy \) represents the data type of the input parameter at index \( pInd. \) Element \( eOp \) represents the expression operator in a condition and \( val \) represents the actual value in the condition. Tuples in list \( conds \) are ordered by the value of element \( pInd. \) In a condition, the data type of a service parameter \((pTy)\) is equal to the data type of the value \( val \) (represented by \( val.type \) in Figure 4.48(a)) in the condition. In a condition, the data type \((pTy)\) of an extra parameter (e.g. two bool parameters in ‘sal’ signature in Figure 4.48(b)) is inferred from the data type of the value \( val \) from the condition.

\[
\begin{align*}
L &= \{} , Z = \{} \\
\text{Where: } cSel = <cInd, conds>, \text{ conds} = [<lOp, pInd, pTy, eOp, val>, ... ] , \text{ conds is a list ordered by the values of element } pInd . pTy = val.type.
\end{align*}
\]

\( \text{Figure 4.48: Example of a selector constraint} \)

In order to illustrate the use of a selector constraint in a practical system, a simple composite (shown in Figure 4.48(b)) of two components \( A \) and \( B \) with same service \( \text{sal} \) (to calculate the salary) is considered. The first input argument is the number of worked hours and the second argument is the pay rate per hour. The difference between the two components is that component \( B \) deducts some fixed amount from the salary for some employee benefits. In a shown \( SEL \) composite, the selection of a component is based on some values not part of input arguments of the service; these arguments are passed by the system. Hence, for the composite’s service \( \text{sal} \), a constraint (shown as input to parser function in Figure 4.48(c)) is added to the connector. A parser function in the connector parses this constraint.

For a service in a selector composite, the execution of a service of one of two components is part of the connector’s semantics. Hence, an FCL constraint’s operational semantics is to evaluate the condition in the component selection expressions (in sequence) in the constraint. The evaluation of a constraint is an identifier for one of two components. The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the selector connector for control/data.
4.5. CONSTRAINED EXOGENOUS CONNECTORS

The selector connector shown in Figure 4.48(b) with a constraint shown in Figure 4.48(c) is defined as a CP-net shown in Figure 4.50. Transitions \( T2-T8 \) in Figure 4.51 define the operational semantics of a selector constraint.

**Interface Generation**

A selector connector (SEL) creates a composite by composing two components. The interface generation semantics of a selector connector with default connector properties (with \( L = {} \)) are shown in Figure 4.49.

![Interface Generation Diagram](image-url)

**Figure 4.49: Interface generation**

For SEL, a composed component is taken as an input of set \( \{cSigs\} \) of provided services and output is a set \( \{pSigs\} \) of provided services for the composite. A member in set \( cSigs \) is described in Section 4.4.2. A service \( \{pSigs\} \) is a 3-tuple of \((sID, sSig, cList)\) service identifier \( sID \), service signature \( sSig \) and a list \( cList \) of composed component indices. Service signature \( sSig \) in \( pSigs \) is described in Section 4.4.1.

The process of interface generation is very simple that creates the interface in two steps described here informally. In the first step, add service signatures in the provided services set (with value 1 in \( cList \)) for all services (with service/parameter annotations) from the first component unconditionally. In the second step, match all services of the second component with the provided services. Two services are matched successfully if their service annotations are same and these services have same lists of input/output parameters. For two parameter list to be same, the order of parameters along with their annotations must be same. For an unmatched service, add a service signature in the provided services set (with value 2 in \( cList \)) along with service/parameter annotations. For an matched service, add value 2 in \( cList \) in the matched provided service. The rigorous process of interface generation is given in Appendix C.2.
As with the guard connector, the interface generation mechanism of the selector connector extends the iList element of a service signature if there are extra parameters (not from the original service signature from the composed components) used in the service constraint (as shown in 4.48(b) for the constraint shown in 4.48(c)). This is the only possible involvement of a constraint in the interface generation mechanism of the selector connector. Unlike the guard connector, the size of parsed set $Z'$ of component selection constraints is not equal to the size of the provided interface $pSigs$.

In the selector composite, for a provided service, the list of input parameters is the union of input arguments (of the default provided) and extra parameters (used in the provided service constraint). The formal process of changing service signatures with extra parameters in the service’s constraint is given in Appendix C.3.

In the deployment phase, a composition connector property for propagated services can be changed by the system designer. Hence, only desirable services can be selected for the provided services set of the composite.

**Control and Data Flow**

The control and data flow of a selector connector is shown in Figure 4.50.

During execution, a service request from the provided interface of a composite is
4.5. CONSTRAINED EXOGENOUS CONNECTORS

accepted by the composition connector. On receiving a request message for a service from outside the connector, the composition connector (SEL1 in Figure 4.50) initiates exactly one service call (from a composed component) with data received in the request message. For Figure 4.50, subpage for Group1 is shown in Figure 4.51. The CP-net of a selector connector has many similarities with a guard’s CP-net shown in Figure 4.40.

Figure 4.51: Group1 of SEL1

Transition T1 along with its associated places Req, Temp, mem and args) of the selector connector is exactly the same as with the guard connector.

Next enabled transition T2 gets the count of components offering the requested service from the requested service’s constraint from place Const and writes that count to place arity. This transition also writes component selection expressions to place cConsts. By reading the component selection expressions from place cConsts, transition T3 writes the first component index (offering the constrained service) to place selC and writes the selection expression for the first component to place sConst. Place
cInd has an initial token with value 1 to represent the first component of a constrained service. These transitions are shown in Figure 4.52.

![Diagram showing transitions T2 and T3 in Selector CP-net](image)

**Figure 4.52: Transitions T2 and T3 in Selector CP-net**

There can be more than one (Boolean) expression in a condition for the selection of a component. Transition T4 passes a pair of the first Boolean expression and the index for the next expression to place aCond. The first element of an expression can be ‘null’ (processed by transition T5), ‘and’ (processed by transition T5’) or ‘or’ (processed by transition T5’’). Transitions T5, T5’ and T5’’ along with their associated places of the selector connector are exactly the same as with transitions T4, T4’ and T4’’ of the guard connector. Hence, we do not show or describe these transitions here. Initially, place result holds an integer value 0 to represent ‘false’ for the selection of a component at the index stored in place selC. Transition T6 sends the intermediate result (true/false) for processing the expressions (by transition T5 or T5’ or T5’’) of a component at an index (stored in place selC) to place result. This transition passes the index for getting the next expression to place nxtCond. All Boolean expressions in place sConst are processed in the explained way. Transitions T4 and T6 are shown in Figure 4.53.

If the final result of component selection at the index from selC is true, then transition T7 fires to write to place Temp2; otherwise, transition T7’ fires to write to place...
4.5. CONSTRAINED EXOGENOUS CONNECTORS

Figure 4.53: Transitions $T4$ and $T6$ in Selector CP-net

$Temp3$ and transition $T8$ fires to increment the component index from place $cInd$. CP-net (from transition $T3$ to transition $T7$ or transition $T7'$) repeats until selection condition for a component is satisfied. These transitions are shown in Figure 4.54.

Figure 4.54: Transitions $T7$, $T7'$ and $T8$ in Selector CP-net

On getting the successful selection conditions from place $Temp2$ for a component in place $cInd$, a transition ($T9$ for the first component or $T9'$ for the second component) to create a service request (for a service from place $CI1$ for the first component or from place $CI2$ for the second component) to a composed component is fired. The service request to a composed component is sent to place $CPo$ for the component and response is received in place $CPi$ for the component. These two transitions are shown in Figure 4.55.

Finally a transition ($T10$ for the first component or $T10'$ for the second component)
Figure 4.55: Transitions $T9$ and $T9'$ in Selector CP-net

creates the response for the requested service to be sent out to the service requester. These two transitions are shown in Figure 4.56.

Figure 4.56: Transitions $T10$ and $T10'$ in Selector CP-net

The description of the simulation (in the CPN tools) of the selector connector (shown in Figure 4.50) is provided in Appendix B.7.

### 4.5.4 Finite Loop Connector

The finite loop (floop) connector is a unary connector with a constraint to terminate making request to the connected component. In a system, for the floop, generation
of the provided interface of an adapted component and the control/data flows to the
connected component are dependent upon the FCL constraints of the connector.

In this section, EBNF for floop constraints are defined/explained. Next, for a floop
connector, semantics of generation of provided interface and flows of control/data are
declared. Operational semantics of selector constraints are part of the control/data flow
semantics of the connector.

**EBNF for FLoop Constraints**

The EBNF grammar rules of FCL constraints for the floop connector are shown in
Figure 4.57. Like the selector and guard connectors, for floop, the EBNF rules for the
non-terminal \langle condition \rangle (shown in Figure 4.37) are same; hence, grammar rules for
\langle condition \rangle are not shown in Figure 4.57.

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;floopConstraint&gt;</td>
<td>sN &quot;{&quot; iterate &lt;num&gt; &quot;times&quot; [&quot;or&quot; &lt;condition&gt;] &quot;}&quot;</td>
</tr>
<tr>
<td>&lt;sN&gt;</td>
<td>ID</td>
</tr>
<tr>
<td>&lt;num&gt;</td>
<td>Num</td>
</tr>
</tbody>
</table>

*Note: sN represents a service name.*

**Figure 4.57: EBNF of FCL constraint for a floop**

With the help of this constraint, the behaviour of a floop connector in an adapted
component is fixed, as described in Section 4.2.3. For a service generated by a floop
connector, an FCL constraint defines the criteria (based on service input arguments)
whether to repeat the service request to the connected component or not. According to
conventional X-MAN semantics, an infinite loop connector at any connector level other
than the root level makes the system meaningless. In order to achieve definite termina-
tion of the finite loop connector, the EBNF defines the must part of the constraint (e.g.
‘iterate 5 times’). With the help of this part of the constraint, floop terminates repeating
service requests after a fixed number of iterations if the optional part of the constraint
(shown in square brackets) does not lead to the loop termination. Unlike EBNF of the
selector and guard connectors, a parameter (e.g. param0) in the Boolean expressions in
a floop condition represents the output parameter of a constrained service; furthermore,
no extra parameters can be added in a service signature by the service’s constraint.

A parser function (Figure 4.58(a)) in the floop connector parses a constraint to a
tuple (representing the abstract syntax tree for the constraint) of a service ID (sID),
and a tuple of set (conds) of conditions (or Boolean expressions) and a positive integer
value \((iVal)\) to repeat forwarding the control/data to the connected component. A floop condition is a 5-tuple \((\langle lOp, pInd, pTy, eOp, val \rangle)\); element \(lOp\) represents a logical operator between two conditions; for the first condition there is “null” value for this element. Element \(pInd\) represents the output parameter index of the service and \(pTy\) represents the data type of \(pInd\). Element \(eOp\) represents the expression operator in a condition and \(val\) represents the actual value used in the condition. Tuples in list \(conds\) are ordered by the value of element \(pInd\). In a condition, the data type of a service parameter \((pTy)\) is equal to the data type of the value \(val\) (represented by \(val.type\) in Figure 4.58(a)) in the condition.

In order to illustrate the use of the floop constraint, a simple adapted component (shown in Figure 4.58(b)) \(A'\) with a service \(func\) is considered. For the first iteration, \(FL1\) forwards the request to the connected component unconditionally. Then \(FL1\) iterates the request to the connected component based on the constraint. Hence, for service \(func\) of component \(A'\), one of the two constraints (shown as input to parser function in Figure 4.58(c)) is added to the connector. A parser function in the connector parses this constraint. In Figure 4.58(c), the first constraint \((func\{iterate 3 times or param0=true\})\) means the iteration of service \(func\) until the first output parameter is not true within total 3 iterations. Second constraint \((func\{iterate 3 times\})\) means the iteration of service \(func\) for three times.
For a service in a floop adapted component, the execution of a service is part of the connector’s semantics. Hence, an FCL constraint’s operational semantics is to evaluate the condition in the constraint. The evaluation of a condition with one or more Boolean expressions is a value from set $T$. The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the floop connector for control/data flows to the connected component.

The floop connector shown in Figure 4.58(b) with a constraint shown in Figure 4.58(c) is defined as a CP-net shown in Figure 4.60. Transitions $T5$-$T9/T9'$ in Figure 4.61 define the operational semantics of a floop constraint.

**Interface Generation**

A floop connector adapts a component; the interface generation semantics of a floop connector with default connector properties (with $L = {}$ and $Z = {}$) is shown as a function in Figure 4.59.

For the floop connector, the set of provided services $cSigs$ of the connected component is the input to the connector and output is a set $pSigs$ of provided services for the adapted component. Members in sets $cSigs$ and $pSigs$ are described in Section 4.4.2. In a floop adapted component, every service in the provided interface must have a constraint. Length of both sets ($cSigs$ and $pSigs$) is equal. The floop connector is a (one-to-one correspondence) function between sets $cSigs$ and $pSigs$.

In the deployment phase, a connector’s property for propagated services can be changed. The default property (with $L = {}$) propagates all services of the connected component. A system designer can change this property by adding only desirable services to be propagated into the set of provided services of the adapted component.
Control and Data Flow

For an example in Figure 4.58, the control/data flow semantics of the floop connector FL1 are shown with the help of a CP-net inside the connector in Figure 4.60.

Figure 4.60: Control and Data Flow of FL1

For Figure 4.60, subpage for Group 1 is shown in Figure 4.61.

Figure 4.61: Group 1 of FL1

Transition T1 along with its associated places Req, Temp and mem) of the floop connector is exactly the same as with the invocation connector.
4.5. CONSTRAINED EXOGENOUS CONNECTORS

Next enabled transition $T_2$ is fired to create a request (in place $Temp_1$) for the reference service from the connected component. This request is sent to place $CPo$ and place $Temp_2$ by transition $T_3$. Next the request is executed by the connected component and a response is submitted to place $CPi$. Transition $T_4$ stores the service ID of the connected component into place $Temp_3$ and passes the result set to place $args$. These three transitions are shown in Figure 4.62.

![Figure 4.62: Transitions T2, T3 and T4 in Floop CP-net](image)

Transition $T_5$ is enabled if a constraint exists in place $Const$ for the requested service. After reading the constraint, $T_5$ stores the count of (Boolean) expressions in the conditional constraint to place $cCount$, expressions to place $sConst$, and $iVal$ to place $fCount$. Transition $T_6$ processes all expressions with Boolean parameters in the constraint’s expressions from place $sConst$ if the value in place $fCount$ is greater than 0. These two transitions are shown in Figure 4.63.

![Figure 4.63: Transitions T5 and T6 in Floop CP-net](image)

For simplicity, unlike the guard and selector connectors, here we include only one transition $T_7$ (counterpart for transition $T_4$ in the guard CP-net and for transition $T_5$
in the selector CP-net) that processes the first Boolean expression from the floop condition. Transition $T_7$ writes a pair of result (0 for false and 1 for true) of a constraint’s condition and an increment of a value (representing the index of an expression from the requested service’s constraint) read from place $index$ (with initial value 0). Transition $T_8$ places the new index value to place $index$ and expression evaluation to place $results$. These two transitions are shown in Figure 4.64.

![Figure 4.64: Transitions $T_7$ and $T_8$ in Floop CP-net](image)

Transition $T_9$ is enabled to activate the next iteration. $T_9$ is enabled if the loop count value in place $fCount$ is greater than 0 and the condition for terminating the loop is not satisfied. Transition $T_{10}$ recreates the request (in place $CPo$) for the service from the connected component. CP-net from transitions $T_3$ to $T_{10}$ executes for $iVal$ times. These two transitions are shown in Figure 4.65.

![Figure 4.65: Transitions $T_9$ and $T_{10}$ in Floop CP-net](image)

Transition $T_{9}'$ is enabled if the result (in place $result$) of the requested service’s constraint’s condition becomes true before the end of loop (value in place $fCount$ becomes 0). $T_{9}'$ sends the result set to place $Temp6$. After completing the loop by decrementing the value in place $fCount$ to 0, transition $T_{10}'$ becomes enabled and
sends the result set to place Temp6. Finally, transition T11 creates the response for the requested service. These three transitions are shown in Figure 4.66.

![Figure 4.66: Transitions T9', T10' and T11 in Floop CP-net](image)

The description of the simulation (in the CPN tools) of the floop connector (shown in Figure 4.60) is provided in Appendix B.8.

## 4.6 The Bank Example

In order to illustrate the behaviour of exogenous connectors, the bank example from Section 4.1.5 is considered with further extension, as shown in Figure 4.67. The bank system is annotated by the component/composite interfaces and FCL constraints for the constrained connectors. Unary exogenous connectors (adaptors) do not change the name of the service of the adapted component. In contrast, compound services of composition connectors can be named meaningfully by the developer.

Sequencer SEQ1 creates a compound service (named as getData) of two sub-services (readCard of CR and readPin of PR). In the signature of the compound service, list of input/output parameters are the concatenation of respective parameters of two sub-services, as described in Section 4.4.3. Pipe PIPE1 composes the composite of SEQ1 with CB; in the signature of PIPE1’s compound service (login), list of output parameters is the concatenation of output parameters of the sub-services (getData of SEQ1 and authorise of CB). However, for the input parameters, list of input parameters
are the concatenation of input parameters of the sub-services excluding parameters for which there are data mappings in PIPE1’s constraint, as described in Section 4.5.1. As both input parameters of authorise service are mapped with the output parameters of the getData service (defined in PIPE1’s constraint), there are no input parameters for the login service. The finite loop connector L1 does not change the service interface of the adapted component, as described in Section 4.5.4.

Selector SEL1 creates a compound service withdraw of two sub-services (one from each composed component) by using our full-signature matching mechanism defined in Figure 4.49. In withdraw service, there are no extra parameters used in the connector’s constraint. For using an extra parameter, guard connectors (G1, G2 and G3) create the service signature different than the corresponding service signature of the
adapted component, as described in Section 4.5.2. Moreover, guard connectors make output parameters of constrained type. PIPE2 composes four components of L1, G1, G2 and G3 to create a composite with one compound service named as cshWithdaw. Constraint for service cshWithdaw defines data mapping for readAmount service of G1, withdraw service of G2 and confiscate service of G3. Finally, the component of PIPE2 is adapted by the infinite loop connector L2. A service created by this connector does not have input or output parameters, as described in Section 4.4.2. In this example, the use of CC component with confiscate service shows one way of handling exceptions in X-MAN'.

For the run-time phase, in the system, numbered request/response messages are shown to represent the control/data flow in the system. All connectors (except the root infinite loop connector) generate a response message for each request message. Hence, a pair of messages represents an interaction between a connector and a component. In the system, an execution cycle starts with request message numbered as ‘1’ and completes with response message numbered as either ‘24’ (a successful transaction with withdrawn money) or ‘18’ (a failed transaction with card confiscated).

### 4.7 Exogenous Composition Framework

To model and simulate X-MAN' systems (with some proposed extensions from Section 4.2.2 and Section 4.2.3), we have developed a prototype tool (Figure 4.68(a)) exogenous composition framework (ECF). The purpose of this tool is to construct a system and to evaluate the correct control/data flows in the system. In this tool, some of our proposed extensions for X-MAN' are not implemented. For example, inferring data types for extra input parameters from the FCL constraints and parsing FCL constraints are not implemented. FCL constraints are parsed manually and added to the exogenous connectors. In ECF, exogenous connectors (described in Section 4.3) are implemented and stored in the connector repository. Using ECF, we construct systems by using encapsulated components and exogenous connectors from the respective repositories. The code for a composite ‘sample’ is shown in the code view of ECF in Figure 4.68(b). During the construction, intermediate partial systems as well as the final system can be simulated with test data. The system’s simulation result in textual form is displayed in the output tab (Figure 4.68(b)). ECF provides special views for component/connector repositories, for the interfaces of selected components and for the generated system code.
4.8 Chapter Summary

In this chapter, we have described the conventional X-MAN component model in necessary details. In order to achieve further support for the incremental construction, a number of extensions to the model have been suggested. Our extended X-MAN is referred to as X-MAN'. For X-MAN' components, our proposed extensions include distinguishing components for user interactions (referred to as UI components) from the components with computational behaviour, introducing annotations for services in a component’s interface, and the constrained output parameters for a service. Distinguishing UI components from the computational components can help divide the system architecture into two parts. Moreover, a system can be adapted by replacing UI components (if needed) without modifying the functional or computational behaviour of the system. For exogenous connectors in X-MAN', our proposed extensions include the concept of service propagation and behaviour containment, open composition connectors and the flow constraint language.

In this chapter, we define the static and dynamic semantics of all exogenous connectors with our proposed extensions. We developed an exogenous composition framework (ECF) that implements the semantics of X-MAN'. Finally, by using a simple bank example, the behaviour of exogenous connectors are illustrated.

In the next chapter, we will present the construction guidelines to achieve incremental construction in X-MAN'.
Chapter 5

Incremental Construction Using X-MAN'

In this chapter, construction guidelines are proposed to construct systems in X-MAN'. In line with the semantics of X-MAN', these guidelines include possible ways to increment a system, ways of refactoring a system for further construction and the construction process. In this chapter, we present the set of all possible increments with respect to our proposed extensions to the X-MAN component model \(^1\). Moreover, for X-MAN' systems, ways to refactor X-MAN' system architecture to support incremental construction are also identified and presented with respect to our proposed extensions. Next, by using the proposed guidelines, an extended bank system (based on the simple bank example from Section 4.6) is constructed. Lastly, the extended bank system is constructed and simulated in ECF.

In CBD, component models define components and composition mechanisms for components. In general, component models do not provide guidelines for incremental construction. However, for incremental construction such guidelines are helpful in using the component model. Hence, having a component model that can be used for incremental construction is only half the story. The other half concerns the construction guidelines which include: (i) possible ways to increment a system, (ii) refactoring steps to resolve the issues that cause an hindrance in the construction process, and (iii) the iterative construction process.

In this chapter, by using the extended X-MAN component model (X-MAN’), we address the aforementioned other half of the story.

\(^1\)This set was identified as part of work carried out jointly with my colleagues Lau, Ng and Tran [62].
5.1 Towards Achieving Incremental Construction in X-MAN′

For system construction, a repository (of pre-developed components with fixed behaviour and a set of exogenous connectors) plays a central role in X-MAN′, as described in Section 4.1.3. In order to focus on incremental construction of systems in the deployment phase, we assume that all components with functional behaviour are available in the repository. Moreover, to keep focus on achieving the functional behaviour incrementally bit by bit, we also assume that components perform the expected computation and the user of the system enters the expected data correctly.

During system construction, a system developer can pause the construction process and switch to the design phase (of the X-MAN′ component life cycle) to develop a required component if the required component is not available in the repository, as described in Section 4.1.3. After developing the component, the construction process can be resumed by switching back to the deployment phase to use the developed component from the repository. As pre-developed components are used in the system construction process, an incremented system during the construction process is comprised of repository components only. Hence, the system construction process is bottom-up and truly CBD [73, 29], as described in Section 3.2. Exception handling can be achieved by adding extra components and/or guard connectors to ensure the correct behaviour, as shown in the bank example in Section 4.6.

5.2 Adding Increments

By incremental construction, we mean to construct a system by adding behavioural increments in many iterations. In Section 3.1.2, we defined the behaviour of a system and a component. In X-MAN′, system architecture is segregated into control and computation layers, as shown in Figure 4.5. In X-MAN′, the behaviour of a system is a set of services exhibited by the system, as described in Section 3.1.1; making a service request from this set results in executing computation(s) (from the computation layer) according to the system’s control flow (defined by the control layer). As a collection of X-MAN′ model elements, a system can be incremented by adding three kinds of element: (i) a component, (ii) a connector, and (iii) a composite. Adding a connector (an increment to the control layer) or adding a component (an increment to the computation layer) or adding a composite (an increment to both layers) means incrementing
5.2. ADDING INCREMENTS

the system’s behaviour.

A system’s behaviour can also be incremented by using function \( apL \) to append a service in set \( L \) of a connector; the appended service is propagated and contained in the system’s behaviour, as described in Section 4.2.3. However, in this way of incrementing system’s behaviour, the system architecture is not incremented as no new model element is added. In this section, by using suitable example(s), we describe different ways of incrementing an X-MAN’ system that increments both the system’s behaviour and its architecture. After adding an increment to a system, provided interfaces of affected composites/components in the system are refreshed (worked out again) to show the changes in their interface.

As the objective of this section is to add increments to a system that increments both the system’s behaviour and its architecture, propagating one or more services of a component/composite is not significant. Hence, for simplicity and without loss of generality, we assume that in a system components have a single exhibited service and connectors propagate all services (i.e. for all connectors \( L=\{\} \), as described in Section 4.3). For example, we reconsider system architecture from Figure 4.5 in which each component has one service and each connector propagates all services, as shown in Figure 5.1.

By virtue of service containment (described in Section 3.1.3), a service created by an exogenous connector contains the service(s) of the connected component(s). Hence, in Figure 5.1, the behaviour of the adopted/composite component contains the
behaviour of the connected component(s). The control layer can be imagined as a composite connector \cite{66} with function \( CL : n \times \ldots n \rightarrow n \), where ‘\( n \)’ represents a service) to create the functional interface of the system (\( B_{P5} \) or \( B_{sys} \)). The exact behaviour of the system is based on the types of connectors and their position in the hierarchy in the control layer.

For a construction step (Figure 3.1) in X-MAN\(^{\prime} \), an increment (\( inc_i \)) is composed with the current system (\( S_i \)) in one of four possible ways, as shown in Figure 5.2(a). The incremented system (\( S_{i+1} \)) must be a legitimate design in X-MAN\(^{\prime} \) and the behaviour of the incremented system must contain the behaviour of the current system (\( B_{S_i} \subseteq B_{S_{i+1}} \)) to comply with the definition of incremental construction (from Section 3.1.1). After adding an increment to a constrained connector in the system, the connector’s constraints may be refined by using respective connector features (to modify/extend a constraint). These features represent two core rules to refine a constraint; however, for clarity, we define different refinement rules specific to exogenous connectors, as shown in Figure 5.2(b).

```
Modify PIPE's constraint
Extend SEL's constraint
Modify SEL's constraint
Modify PIPE's constraint
Modify adaptor's constraint
```

Rule 1  Modify SEL's constraint
Rule 2  Modify PIPE's constraint
Rule 3  Modify adaptor's constraint
Rule 4  Extend SEL's constraint
Rule 5  Modify PIPE's constraint

(a)  (b)

Figure 5.2: Adding increments to a system

For incremental construction, after adding increments and refining connector constraints, the behaviour of the existing system (\( S_i \)) and of increments are contained by the behaviour of the incremented system (\( S_{i+1} \)), as described in Section 3.1.3. Moreover, test cases of \( S_i \) can be used to create test cases for \( S_{i+1} \), as described in Section 3.1.3.

5.2.1 By Composition

The composition mechanism in X-MAN\(^{\prime} \) is control coordination, as described in Section 2.4.1. An exogenous composition connector composes at least two components
to create a composite. In order to construct a system incrementally, we assume that the starting system \( (S_0) \) is a single component and the behaviour of the system is the behaviour of the starting component.

Increments can be added to the control and computation layers in the current system by using the composition mechanisms, i.e., a composition connector \( K_i \) and a component (or a composite) \( C_i \) to the current system \( S_i \) can be added, and thus increment the behaviour of \( S_i \) by adding extra control (\( K_i \) and connectors in the composite \( C_i \)) and computation of the component (or components in \( C_i \)). This can be done in two ways:

(i) Adding the increments \( K_i \) and \( C_i \) to \( S_i \) itself, i.e. composing \( S_i \) with \( C_i \) using \( K_i \).

(ii) Adding the increments \( K_i \) and \( C_i \) to a component/composite in \( S_i \).

In these two alternatives, for incremental construction, increments can be added to atomic and composite components alike. Hence, these mechanisms can be applied to an initial system \( S_0 \) that is just an atomic component, and to any subsequent \( S_i \) that is a composite component. Because of the hierarchical nature of composition in X-MAN’, these mechanisms define incremental construction in a recursive manner for any \( S_i \).

In the following subsections, with the help of examples, we will describe the aforementioned two ways of adding increments to a system.

**Adding Increments to the System**

In Figure 5.3, a system \( (S_0) \) with a single component \( A \) is assumed. The behaviour of a component is a set of services exhibited by the component, as described in Section 3.1.2. \( S_0 \) can be incremented by adding a component by composition, as shown in Figure 5.3(a). Relative to \( S_0 \), component \( B \) and the composition connector \( P1 \) together form an increment (shown as \( inc_0 \)). The component \( B \) is an increment in the computation layer and the composition connector \( P1 \) is an increment in the control layer of the X-MAN’ system. In this way of incrementing the system, the system is composed with a component \( B \) by a composition connector \( P1 \).

In order to satisfy the behaviour containment property for incremental construction, the behaviour of the incremented system (service ‘ab’ of \( S_1 \)) must contain the behaviour of the current system (service ‘a’ of \( S_0 \) or component \( A \)). As per our assumption that all services are propagated, the behaviour of \( S_1 \) contains the behaviour of \( S_0 \). However, the exact behaviour of \( S_1 \)’s service ‘ab’ is based on the types of connectors and their hierarchy in the control layer, that is a single connector \( P1 \) in
Similarly a composite increment can be added to a system by composition, as shown in Figure 5.3(b). In this example, a composite of connector P2 is composed with the current system by a composition connector P1. The composite of connector P2 and the composition connector P1 together is an increment to the system (shown as $inc_0$). In the system, components $B$ and $C$ are increments to the computation layer, and composition connectors P1 and P2 are increments to the control layer. The behaviour of the incremented system ($S_1$) contains the behaviour of current system ($S_0$).

In the same way, an incremented system (a composite) can be incremented by composition with a single component or with a composite. For example, we consider the system in Figure 5.3(a) as the current system (a composite of connector P1) and increment the system, as shown in Figure 5.3(c). In this increment, the current system is composed with the composite of connector P2 by a composition connector P3. The composition connector P3 and the added composite of connector P2 together is an increment (shown as $inc_1$) to the current system; the behaviour of system $S_2$ contains the behaviour of system $S_1$. In each increment (in Figure 5.3), the composition connector is part of the increment.

**Adding Increments to a Sub-Component**

In the system, an exogenous composition connector represents a composite component (or simply a composite). Adding an increment to a sub-component means adding an increment to an atomic component or to a composite in the system. In Figure 5.4(a), we assume a system ($S_0$) with two components $A$ and $B$ composed by a connector P1.

This system can be incremented by adding a component to the sub-component...
5.2. ADDING INCREMENTS

Figure 5.4: Increments to the sub-component by composition

(component B) by composition, as shown in Figure 5.4(b). In this increment, component B of the current system is composed with component C by a connector P2. The added component C and the composition connector P2 together is an increment (shown as inc0) to the system. In the system, component C is an increment to the computation layer and the composition connector P2 is an increment to the control layer. The behaviour of the incremented system (S1) contains the behaviour of the current system (S0).

To a sub-component, an increment can also be a composite, as shown in Figure 5.4(c). In this example, a composite of connector P3 is composed with sub-component B by a composition connector P2. The composite of connector P3 and the composition connector P2 together is an increment (shown as inc0) to the current system (S0). To the system, component C and D are increments to the computation layer and connectors P2 and P3 are increments to the control layer. The behaviour of the incremented system (S1) contains the behaviour of the current system (S0).

Rules to Refine Constraints

In the previous subsection, we discussed the general cases of adding increments to a sub-component by composition. Because of the open composition connectors in X-MAN’, an increment can be composed with an existing composition connector by increasing the connector arity. Hence, in Figure 5.4, connectors P1 (connector of the sub-component) and P2 (root connector of the increment) must be of different types.
and for incremental construction the behaviour of the incremented system $S_1$ must contain the behaviour of the previous system. In the context of refining connector constraints, there are five (3+2) possible cases for incrementing a system by adding an increment to a sub-component, as shown in Figure 5.5. For uniformity, the component connected to an adaptor connector in also considered as the sub-component of the adaptor connector.

Figure 5.5: Possible cases of adding increments to a sub-component by composition

In the first three cases P1 is a composition connector and in the other two cases P1 is an adaptor. Unlike adding increments to the system, for adding increments to a sub-component, the constraints of the composing connector (P1 in Figure 5.4) must be refined if the composing connector is a constrained connector. Hence, the effects of the increment to the sub-component must be analysed carefully with respect to connectors P1 and P2 in Figure 5.4. This analysis can lead towards defining rules to refine (modify/extend) the constraints of a constrained connector if an increment is added to a sub-component of the connector.

Figure 5.5(a) represents a case of P1 as a sequencer connector. As sequencer is an unconstrained connector, adding an increment with a pipe or a selector as P2 is straightforward because no constraint refinement is required. In Figure 5.5(b), connector P1 is a selector which uses the full-signature matching mechanism (described in Section 4.5.3) to match services from the composed components and has a constraint to select a matched service from one component. Adding an increment (with two possible cases when P2 is a pipe or a sequencer connector) to $S_0$ requires P1’s constraint
to be modified. Here, we define a generic rule to modify a selector’s constraint.

**[Rule 1]** Modify SEL’s constraint. After adding an increment to a selector’s composite, the selector’s constraint is modified to handle the coordination for the added increment. This rule can be defined as a generic deployment phase service (\(\text{modConst}(sN,nConst)\)), where \(sN\) represents the compound service name and \(nConst\) represents the new constraint to override the previous constraint).

After adding an increment to a sub-component of a selector connector, the sub-component’s ID in the connector’s constraint has to be replaced with the added component’s ID. In order to achieve this change by applying Rule 1, an overloaded service \(\text{modConst} (\text{\textquoteright}modConst(sN,oID,nID)\text{\textquoteright})\), where \(sN\) represents the compound service name, \(oID\) represents the sub-component’s ID in the current system and \(nID\) represents the added increment’s ID) can be defined. After adding an increment to \(S_0\) (in Figure 5.6(a)), the application of Rule 1 refines SEL.1’s constraint in the incremented system \(S_1\) as shown in Figure 5.6(b).

![Figure 5.6: Application of Rule 1](image)

A pipe is a sub-type of the sequencer connector (described in Section 4.2.3) and the constraint for the pipe (described in Section 4.5.1) does not necessarily have to pass data values between all sub-services of a compound service. Hence for P1 as pipe, there is no case to increment the system where P2 is a sequencer connector because the desired behaviour can be achieved with P1 alone. Considering P1 as a pipe connector, there is only one case and that is with P2 as a selector connector. In this case (Figure 5.5(c)), after adding an increment to a sub-component of a pipe connector, the
constraint of the pipe connector has to be modified for data passing to the added composite (of SEL1). As described in Section 4.5.1, the constraint of a pipe connector is component and service specific between the source and the target of data flow.

[Rule 2] Modify PIPE’s constraint. After adding an increment to a sub-component of a pipe connector, the data flow expression for the previous sub-component in the pipe connector may need to be modified to target a service of the new composite (including the sub-component). The name of the new compound service or the identifiers of the source/target components may need to be modified. Moreover, the sub-services of the compound service may need to be modified. As with Rule 1, in the pipe connector, Rule 2 can be defined as a deployment phase service (\(modConst(sN,oID,dfExp)\)), where \(sN\) represents the compound service name, \(oID\) represents the sub-component’s ID in the current system and \(dfExp\) represents the data flow expressions for the added component in the increment). After adding an increment to \(S_0\) (in Figure 5.7(a)), the application of Rule 2 refines PIPE1’s constraint in the incremented system \(S_1\) as shown in Figure 5.7(b).

![Figure 5.7: Application of Rule 2](image)

Considering P1 as an adaptor connector (guard or finite loop), there is only one general case with P2 as a composition connector (Figure 5.5 (d) & (e)). Adding an increment to the sub-component of an adaptor by composition may require the constraint of the adaptor to be modified. Here, a rule is defined to modify the adaptor’s constraint.

[Rule 3] Modify adaptor’s constraint. After adding an increment to a sub-component of an adaptor connector, the selection or repetition constraints in the connector may
be modified. The name of the new adapted service, number of parameters and the selection/repetition condition for the service may be modified. For the two adaptor cases, this rule can be defined as deployment phase service (‘modConst(sN,nConst)’, where sN represents the adapted service name and the second parameter represents the modified constraint) to the adaptor connectors. Parameter nConst overrides the existing constraint of a service. After adding an increment to $S_0$ (in Figure 5.8(a)), the application of Rule 3 refines P1’s constraint in the incremented system $S_1$ as shown in Figure 5.8(b).

The Bank Example

Next, by considering a practical but simple bank system to serve clients of two banks, we illustrate adding increments to a system by composition; the system serves one client at a time. By using the incremental construction process from Figure 3.1(a), the system construction starts with an ATM component (Figure 5.9(a)).

The ATM component ($S_0$) accepts a bank card and a pin code to get authentication from the client’s bank system. After the approval, the ATM component accepts an amount to withdraw cash. In the next construction step (Figure 5.9(b)), an increment $inc_0$ (the bank component Bank1 and the composition connector PIPE1) is added by composition to the current system. Incrementing a system this way is an example of adding an increment to the system by composition. PIPE1’s constraint passes the account number and amount to withdraw from the request service of ATM to the withdraw service of Bank1. The behaviour of the incremented system ($S_1$) contains the
CHAPTER 5. INCREMENTAL CONSTRUCTION USING X-MAN

For the next incremental step, the composite shown in Figure 5.9(b) is the current system. An increment is added to the system by composing Bank2 with the component Bank1 by using a selector connector, as shown in Figure 5.9(c). The component Bank2 and the composition connector SEL1 together is the increment (shown as $inc_1$) to the current system. With this increment, the behaviour of the selector connector is to forward the client’s request to one bank component selected by the selector, as defined by the constraint of the connector. The incremented system’s behaviour (to serve the clients of two banks) contains the behaviour of the current system’s behaviour (to serve the clients of Bank1). Using Rule 2, PIPE1’s constraint is modified to pass the data to the service of the selector composite rather than to Bank1’s service, as shown in bold. Incrementing a system this way is an example of adding increment to a sub-component Bank1 by composition.

System $S_2$ can be constructed from the system $S_0$ in a single step by considering the SEL1 composite as an increment and by composing it to the ATM component with a pipe connector. This way of adding an increment would be referred to as adding to the system by composition.
5.2.2 By Increasing the Connector Arity

Exogenous composition connectors have the property of allowing adding increments by increasing the connector arity, as described in Section 4.2.3. Amongst these connectors, the sequencer and pipe connectors define an ordered control (and data) flow between the composed components. In contrast, the selector connector picks only one component to transfer control to. Next, we analyse exogenous composition connectors to explore the meaning of composing increments by increasing the connector arity.

In a composite of two components \( A \) and \( B \), the connector’s arity can be incremented by adding an increment (a component/composite) in three possible ways: (i) add the increment to the left most position with the connector (Figure 5.10(b)), (ii) add the increment in between two components composed by the connector (Figure 5.10(c)), and (iii) add the increment to the right most position with the connector (Figure 5.10(d)).

![Diagram showing incrementation methods](image)

Figure 5.10: By increasing composition connector arity

By incrementing a system in this way, a component (component \( C \) in Figure 5.10) is added to the computation layer of the system, and a new control flow for the added component is added to the control layer (or to connector P1 in Figure 5.10). Hence, the system’s behaviour is incremented. In Figure 5.10, increments are shown in the dashed areas; an increment includes a component and the composition connector whose arity is incremented.

As with adding increments by composition to a sub-component (Figure 5.5) described in Section 5.2.1, the constraint of a connector (pipe and selector) may need to be refined after adding increments by increasing the connector arity.

**By Increasing Selector Arity**

In a system, for the control flow, a selector connector selects only one of the composed components. Supposing the connector P1 in the system (Figure 5.10(a)) is a selector,
the three cases (Figure 5.10(b)-(d)) are equivalent because the selector selects only one component irrespective of its position in the composite. Similar is the case if the increment is a composite increment.

**Rule to Refine Selector Constraint**

After adding an increment to a selector by increasing the arity of the connector, the connector's constraint may be modified (by Rule 1) and/or extended. Here, a rule to extend a selector's constraint is defined.

[Rule 4] **Extend SEL’s Constraint.** After adding an increment to a selector by increasing the connector's arity, the connector's selection condition is extended to select the added component. This rule is defined by a procedure ‘extConst(sN,slCond)’, where ‘sN’ represents the service name and the second parameter is the selection condition for the added component.

**A Payment System**

In order to illustrate adding increments to a selector composite in a system, a simple hypothetical payment system is considered. The payment system can be composed into any system that requires to process customer payments in more than one way. The purpose of this example is to show how components may be added to a system by increasing a selector’s arity.

In Figure 5.11, for a self-service point of sale system in a supermarket, a payment sub-system is constructed incrementally by using the SEL connector. Using the self-service point of sale, a customer is able to pay for the purchased items in many different ways.

The initial system $S_0$ (Figure 5.11(a)) is a composite of two components $A$ and $B$ by SEL1. Component $A$ exhibits a service to accept payments in the form of paper currency and component $B$ has a service to accept payments in the form of coins. For each payment, the selector in the composite selects either component $A$ (if the paper currency is inserted in the system) or component $B$ (if coins are inserted in the system). The selector constraint in system $S_0$ is shown. As described in Section 4.5.3, extra parameters from the selector constraint (e.g. param1) will be appended to the list of input parameters for the compound service signature.

To process a payment from a (debit or credit) card, component $C$ is added to
5.2. ADDING INCREMENTS

$S_0$ by increasing the arity of the SEL1 connector. The incremented system $S_1$ (Figure 5.11(b)) can accept payments in three ways. Using Rule 4, SEL1’s constraint is extended to select component $C$. The extension to select the incremented component is shown in bold.

Next, an increment component $D$, with a service to accept payment in the form of discount or deal offer vouchers, is added to $S_1$ by increasing the arity of SEL1, as shown in Figure 5.11(c). The connector’s constraint is extended (by Rule 4) to add selection condition for component $D$ (shown in bold). In order to illustrate the application of Rule 1 (by using $modConst(sN,nConst)$), the selection criteria is now modified by adding a new parameter in the selector’s constraint to select $C$.

In the selector constraint, conditions to select a component must be disjoint. For a selector composite, the order of connection of the composed components is not significant. Adding component $C$ or component $D$ between two components (on the right of component $A$ and on the left of component $B$) would create an equivalent composite.

**By Increasing Sequencer Arity**

In a system, a sequencer transfers control to the leftmost component and then to the next one on the right and then so on. Now we consider the connector P1 in Figure 5.10(a) as a sequencer; all three cases (Figure 5.10(b)-(d)) are distinct.
A Calculator

We use an example of a calculator to illustrate adding increments to a sequencer composite in a system. In this example, a simple hypothetical calculator is constructed in three steps from a composite of two components, as shown in Figure 5.12(a). Increments are added by increasing the arity of the connector.

<table>
<thead>
<tr>
<th>Component</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>i square(i)</td>
</tr>
<tr>
<td>B</td>
<td>i cube(i)</td>
</tr>
<tr>
<td>C</td>
<td>d add(d,d)</td>
</tr>
<tr>
<td>D</td>
<td>s sub(s,s)</td>
</tr>
<tr>
<td>E</td>
<td>l mul(l,l)</td>
</tr>
<tr>
<td>S_0</td>
<td>(i,i) sc(i,i)</td>
</tr>
<tr>
<td>S_1</td>
<td>(d,i,i) asc(d,d,i,i)</td>
</tr>
<tr>
<td>S_2</td>
<td>(d,i,i,s) ascs(d,d,i,i,s,s)</td>
</tr>
<tr>
<td>S_3</td>
<td>(d,i,i,l,s) ascms(d,d,i,i,l,l,s,s)</td>
</tr>
</tbody>
</table>

where: i=int, d=double, s=short and l=long.

Figure 5.12: Incremental construction of a calculator

In Figure 5.12, after each increment, the system’s interface is created, as described in Section 4.4.3. Java primitive data types (int, short, long, and double) are used for service input/output parameters. The behaviour of the incremented system \(S_{i+1}\) contains the behaviour of the previous system \(S_i\). In the final system \(S_3\), the control flow is shown with the help of numbered request/response messages on the interfaces of the system and sub-systems. A request for service ‘ascms’ creates sequential requests to its sub-services: (i) service ‘add’ of component \(C\), (ii) service ‘square’ of component \(A\), (iii) service ‘cube’ of component \(B\), (iv) service ‘mul’ of component \(E\) and (v) service ‘sub’ of component \(D\).

A Coffee Dispenser

In the calculator example, the sequence of sub-service executions does not suggest a practical system. In order to illustrate the meaningful or more practical use of the sequencer connector, a coffee dispenser system is assumed to correspond to the calculator system \(S_3\) (Figure 5.12(d)). The coffee dispenser component is shown in Figure 5.13 with one service to dispense white coffee with sugar.
5.2. ADDING INCREMENTS

By Increasing Pipe Arity

As with sequencer, a pipe connector transfers control to the leftmost component and then to the next one on the right and then so on. Now we consider the connector P1 in Figure 5.10(a) as a pipe; all three cases (Figure 5.10(b)-(d)) are distinct. Moreover, execution results may flow from a producer component (e.g. component A) to the consumer components on its right (e.g. component B).

Rule to Refine Pipe Constraint

In the pipe connector, there are data flows between the composed components, determined by the connector’s constraint. After adding an increment to a pipe by increasing the arity of the connector, the connector’s constraint may be extended if the newly added component needs to receive result data from the services of other components. Here, a rule is defined to extend a pipe’s constraint with the data flow required for the newly added component.

[Rule 5] Modify PIPE’s constraint. After adding an increment to a pipe by increasing the connector’s arity, the list of data flow expressions for a compound service may be modified to add the data flow for the newly added component. As with other rules, this rule can also be defined as a pipe deployment phase service (‘extConst(sID,dfExp)’; where ‘sID’ represents a compound service ID and ‘dfExp’ represents a data flow expression to be added in the pipe’s constraint). This rule can be used to extend the pipe’s constraints to add non-existing data flow expressions for existing/non-existing sub-services for compound services. Moreover, this rule can be used to add constraints for new compound services.
Expression Calculator

In order to illustrate adding increments to a pipe by increasing arity, a mathematical expression calculator is constructed in three steps in Figure 5.14.

Construction starts with a composite $S_0$ (Figure 5.14(a)); component $A$ has a service to round a real value (e.g. 5.9) and component $B$ has a service to add two integers. As defined by the constraint for the connector, the composite provides a single service which accepts two numbers to add the rounded value of the first number with the second number; the result of the addition is returned by the service.

An increment (component $C$ with a service to square a number) is added to the composite $S_0$ by increasing the arity of the pipe, as shown in Figure 5.14(b). In accordance with Rule 5, the extension to the connector’s constraint is shown in bold. A new data mapping is added in the constraint. The incremented system $S_1$ exhibits a service which accepts two numbers. The first number is squared by component $C$, then the squared value is rounded by component $A$ and then the rounded value is added with the second number by component $B$. The behaviour of $S_1$ contains the behaviours of

Figure 5.14: An expression calculator

<table>
<thead>
<tr>
<th>Component</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>i round(d)</td>
</tr>
<tr>
<td>B</td>
<td>i add(i,i)</td>
</tr>
<tr>
<td>C</td>
<td>d square(d)</td>
</tr>
<tr>
<td>D</td>
<td>print(i)</td>
</tr>
<tr>
<td>E</td>
<td>i mul(i,i)</td>
</tr>
<tr>
<td>$S_0$</td>
<td>(i,i) ra(d,i)</td>
</tr>
<tr>
<td>$S_1$</td>
<td>(d,i,i) sra(d,i)</td>
</tr>
<tr>
<td>$S_2$</td>
<td>(d,i,i) srap(d,i)</td>
</tr>
<tr>
<td>$S_3$</td>
<td>(d,i,i,i) sramp(d,i)</td>
</tr>
</tbody>
</table>

where: i=int and d=double.
5.2. **ADDING INCREMENTS**

$S_0$ and the increment component $C$.

An increment (component $D$ with a service to display a number on a visual display unit) is added to the pipe on the right of component $B$ by increasing the arity of the connector. The incremented system (Figure 5.14(c)) exhibits one service which accepts two numbers as input. Using Rule 5, the extension to the connector’s constraint is shown in bold. The first number is squared by component $C$, then the squared value is rounded by component $A$ and then the rounded value is added with the second number by component $B$. The result of addition from component $B$ is then displayed on a visual display unit by the service of component $D$.

Lastly, component $E$ (with a service to multiply two numbers) is added to the composite between two components, as shown in Figure 5.14(d). The incremented system $S_3$’s behaviour contains the behaviours of $S_2$ and component $E$. Next, by using Rule 5, a data flow expression for an existing service ‘add’ is added; first output value of service ‘round’ (of component $A$) is passed to both input parameters of $B$’s service. The constraint is also modified by using Rule 2 to display the result of ‘mul’ of $E$ by component $D$. Moreover, using Rule 5, a new data mapping for $E$’s non-existing service is added to the pipe’s constraint. Composite $S_3$ has a service to accept two numbers. The first number is squared by component $C$, then the squared value is rounded by component $A$ and then the rounded value is doubled by component $B$. The result of $B$ is then multiplied with the second input number by component $E$. The final result of multiplication by component $E$ is then displayed on a visual display unit by the service of component $D$.

**The Bank Example**

We have already given examples to add increments to a system by increasing the arity of composition connectors. Here, in order to illustrate the system increment by increasing the arity of a composition connector, we consider the bank system from Figure 5.9(c) as the initial system $S_0$ shown in Figure 5.15(a).

A component Bank3 is added to the selector composite by incrementing the arity of the connector and the constraint of the selector is extended by using Rule 4. Doing so adds control (select option for Bank3) and computation (Bank3) to the system (shown as $inc_0$). After the increment, the selector connector can transfer control to any one of the bank components. The behaviour of the incremented system $S_1$ contains the behaviour of the previous system $S_0$. 

5.2.3 By Adding Adaptors

The behaviour of a system can also be incremented by adding adaptors in the control layer. A sub-component of the system can be adapted either by a guard connector or by a finite loop connector. However, the system can be adapted by an infinite loop connector. In Figure 5.16(a), a system of three components composed by a connector is shown. Assuming the connector P1 is a sequencer, three adaptors are added in the system, as shown in Figure 5.16(b).

The guard adaptor (shown as $inc_0$) adapts component $C$ in the system; having this guard, the system can perform two behaviours: (i) a behaviour which invokes a service of $C$ (if the guard condition is true) and (ii) a behaviour which does not invoke a service of $C$ (if the guard condition is false). Two loop adaptors are added to the system as shown in Figure 5.16(b). The loop connector adapting component $A$ (shown as $inc_1$) is a finite loop. A finite loop connector adds finite repetition of the behaviour of the adapted component. A system can be adapted by an infinite loop connector to add infinite repetition of the behaviour of the system. An infinite loop adaptor is added to the system when the system is complete. In a system with infinite loop connector as the root connector, users’ interact with the system by UI components, as described in Section 4.2.2.
5.2. ADDING INCREMENTS

Considering P1 (in $S_0$) as a sequencer, service ‘q’ invokes a service from each component in sequence. Service ‘q’ in $S_1$ invokes a service from each component (at least once) in sequence if the guard condition is satisfied; this is the behaviour of $S_0$. Hence, the behaviour of the incremented system $S_1$ contains the behaviour of the initial system $S_0$. Alternatively, service ‘q’ in $S_1$ invokes services from two components $A$ and $B$ if the guard condition is not satisfied. Once control is transferred to the finite loop connector, the connector repeats making request to component $A$ for a finite number of times. The infinite loop connector, which is only allowed as a top-level connector in a system, makes the system ready to serve more computational requests.

The Bank Example

We use the simplified bank system $S_0$ from Figure 4.67 is to illustrate incrementing by adding adaptors, as shown in Figure 5.17.

The system simply reads a customer’s card and pin code and then gets authentication from the central bank component and then asks for amount to withdraw. PIPE2 acts as a sequencer as it does not have any constraint. The system is incremented in three iterations to add three adaptors. After adding the second increment (guard adaptor), the PIPE2 connector is constrained to pass the authentication response from $L1$ adapted component as input to the adapted component $G1$. The incremented system iterates three times to read the correct pin code. PIPE2 passes the response of authentication result to $G1$; $G1$ passes control to component RA if the authentication is successful. After the completion of one execution cycle, the root connector $L2$ passes
5.2.4 Rule to Refine Adaptor Constraint

After adding an increment to a system, there may be a situation when an adaptor’s constraint is required to be extended to achieve a functional requirement. For example, to construct the bank example (Figure 4.67), we assume that there is a requirement demanding to iterate a behaviour to read a pin code for the inserted card until the pin code is accepted. Later, a new requirement arises to confiscate the card after three unsuccessful iterations; the finite loop adaptor’s constraint is required to be extended. Similarly, in a system, a guard adaptor may need to be extended. In order to avoid to have too many rules, for adaptors, we use Rule 3 to extend an adaptor constraint. In Rule 3, the argument $nConst$ (representing the extended constraint) passed to the function $modConst$ can replace an adaptor constraint.
5.3 Refactoring

In the context of improving the design of an existing system’s (or program’s) code, refactoring is defined as changing a program’s internal structure without changing the program’s external behaviour [41]. Refactoring does not change the behaviour of the program; in other words, refactoring is behaviour preserving. In a system construction, restructuring the system program can make changes that will make evolution of the system easier (or possible). Architecture refactoring can be useful in iterative design of an application [84].

In work about constructing a component-based system by using incremental composition [63], the need for architecture refactoring which can be used to remove redundancy from the system was realised. In this work, in order to merge two system architectures with redundancy (i.e. both having the same component) or to replace a set of connectors with a single connector (e.g. a number of guard connectors with conditions that check the same value are replaced by a selector connector), the system architecture was refactored.

In contrast to the aforementioned general uses of architecture refactoring, in this thesis, we focus on architecture refactoring that is needed to achieve incremental construction in X-MAN'. A system is required to be refactored when adding increments to a system is otherwise not possible or at least not easy. This means that refactoring the system will support the process of incremental construction. However, refactoring will only change a system’s architecture and not its behaviour. Refactoring a system to support incremental construction means refactoring the control layer of the system.

In this section, we identify two ways to refactor the system’s architecture. In these architecture refactorings, a composition connector is replaced by one or more connectors.

5.3.1 Replacing Sequencer by Pipe

Having the same control flow, the pipe connector is a sub-type of the sequencer connector, as described in Section 4.2.3. In a system, if needed, a sequencer connector can be replaced by a pipe connector but with the sequencer’s set of propagated services (L ). We illustrate replacing a sequencer with a pipe in Figure 5.18.

In Figure 5.18(a), a system $S_0$ needs to be incremented with component $F$ such that $F$ will get control after component $D$ and before component $E$ and execution results of component $C$ will be passed as input to component $F$. Here is a situation where adding
component $F$ to the system (shown in Figure 5.18(a)) is not possible. The sequencer connector needs to be replaced with a pipe connector without destroying the existing behaviour. First the connector is replaced (shown in Figure 5.18(b)) then component $F$ is added by increasing the arity (described in Section 5.2.2) of the replaced pipe connector, as shown in Figure 5.18(c).

### 5.3.2 Replacing a Connector by Many Connectors

In X-MAN', while constructing a system, there are situations where we need to replace a composition connector (with arity $> 2$) with several connectors of the same type. Semantically, such connector replacement can also be thought of as connector decomposition. This replacement is behaviour preserving, i.e. the system’s behaviour before the replacement and after the replacement is the same. Being able to do so allows more increments to be added to the system for incremental construction.

Before showing the use of a connector replacement of type $T$ with many connectors of the same type, in order to present the connector replacement clearly, a model transformation approach is adopted. A model transformation is a program that defines how to transform a model conforming to a meta model into another model conforming to the same or different meta model [31]. For a connector replacement in an X-MAN’ system, at an abstract level, the basic concept of the model transformation is shown in Figure 5.19.

In Figure 5.19, the input ($S$) and output ($S'$) models conform to the same meta model (X-MAN’) and the arbitrary input ($I$) from a developer plays a key role in the model transformation. The developer’s input is a tuple $\langle ECon, nList \rangle$ of a connector

![Figure 5.18: Replacing sequencer by pipe](image)
5.3. REFACTORYING

ExCon that needs to be transformed and a nested list nList of components composed by the connector. Transformation definitions includes details necessary to transform a system model by following the X-MAN′ model. In I, the first element (a connector) is replaced by a root and one or more child connectors. The root connector’s arity is equal to the length of a recursive list (second element in I). A list of two or more elements (list/component) represents an exogenous connector.

We will now illustrate how these transformations work, with examples. Two X-MAN′ systems S₀ (with one connector and four components) and S₁ (with two connectors and four components) are shown in Figure 5.20(a) and in Figure 5.20(d) respectively.

Model transformation of system S₀ is shown in Figure 5.20(b) and Figure 5.20(c). Depending on the developer’s input (I), the composition connector P₁ can be replaced with more than one connector of the same type in many different ways. In Figure 5.20(b), determined by input I, connector P₁ of S₀ is replaced with a root connector P₂ of arity two to compose component A and a child connector P₃ of arity three; connector P₃ composes components B, C and D. In Figure 5.20(c), determined by input I, connector P₁ of S₀ is replaced with a root connector P₄ of arity two to compose two connectors, P₅ and P₆. Connector P₅ composes components A and B, and connector P₆ composes components C and D. Similarly, system S₁ in Figure 5.20(d) is transformed to replace P₇ with P₉ and P₁₀ as shown in Figure 5.20(e).

For constrained connectors pipe and selector, the model transformation engine in Figure 5.19 decomposes the constraint of the connector into the constraints for the
replaced connectors such that the behaviour of the system is preserved. A high-level pseudo code outline for the model transformation engine (Figure 5.19) is shown in Figure 5.21. In order to decompose a constraint into more than one constraint, human (developer) guidance is required.

After replacing a connector with several connectors of the same type, an increment can sometimes be added to the system which was not easy or possible without the connector replacement. For example, adding an adaptor connector to adapt the composite of connector P3 becomes possible after the connector replacement shown in Figure 5.20(b).

Similarly, a guard connector can be replaced with several guard connectors. In such a replacement, a connector with a composite condition will be replaced into connectors with atomic conditions (a sub-condition of the composite condition of the original connector).

A (composite) connector (design pattern in [66]) can also be replaced by a number of basic connectors. In a system, as the purpose of refactoring is to enable adding more
5.3. REFACTORING

```plaintext
Procedure tran(model m, Connector c, list L): model
  Begin
  ... If(isSelector(c))
  ... m1 = decomposeSELConstraint(m, c, L);
  ... Else If(isPipe(c))
  ... m1 = decomposePIPEConstraint(m, c, L);
  ... End If
  ... End
```

Figure 5.21: Pseudo code outline of model transformation

increments, we only need to replace or decompose a connector to enable incremental construction.

The Bank Example

We consider the bank example from Figure 4.7 with three banks instead of two banks as our current system, as shown in Figure 5.22(a). The selector connector forwards the request to one of the banks by checking ‘bank id’ from the request. We make a trivial assumption that bank2 and bank3 have a strict policy for its customers to use the ATM system. These two banks require their customers to enter another number (or a thumb expression on a biometric ATM) after entering the pin code for the client’s ATM card. This extra number must be part of every transaction request to the bank. In the current system, it is not possible to add such an increment without destroying the behaviour of the system.

In order to increment the system, first we replace SEL1 from Figure 5.22(a) with two selector connectors, as shown in Figure 5.22(b). The constraint of SEL1 is decomposed into the constraints for SEL2 and SEL3. Doing this replacement, the system behaviour is preserved. Next, we can increment the system by adding an increment $inc_0$ (not shown) to read another number (or thumb expression) for the clients of bank2 and bank3.
5.4 The Construction Process

In CBD, the component building process is distinguished from the system construction process [52, 29, 70]. In this section, for system construction, we assume functional requirements for the system under construction are defined by use cases. If the functional requirements are not stated as use cases, then first requirements are to be processed and written in use cases. For incremental construction, our focus is on the system construction/development process, which is about reusing existing components. Hence, similar to the approach presented in [63], the construction process that we adopt is defined in Figure 5.23. We show the process by using flow chart notations.

Clearly this process is guided by the system designer to identify the existing components from the component repository and to identify the appropriate connectors from the connector repository. In other words, human guidance is needed to select an initial system $S_0$ as well as every increment $inc_i$ to be added at every $i$-th iteration. Moreover, refactoring the system architecture is also human guided. Constructing the target
5.4. **THE CONSTRUCTION PROCESS**

The iterative construction process that we adopt (Figure 5.23) is also written in pseudo code, as shown in Figure 5.24.

The process starts with a use case (the first use case), we identify the initial system (with a functional behaviour) from the use case. In X-MAN', an encapsulated component (atomic or composite) has functional behaviour, therefore the initial system ($S_0$) from the first use case is a component. Next, the iterative process begins.

After having picked the initial system component from the first use case, in the iterative process, we identify one increment at a time and add it to the current system to create an incremented system. The identified increment is added to the current system by one of the possible ways to increment the system, as described in Section 5.2. After adding each increment to the current system, the process checks if there are any more increments to be added to the current system. If we are unable to add an increment to the current system, the current system can go through the refactoring activity which modifies the system (without any change in the behaviour) to support the incremental construction, as described in Section 5.3. The process selects the next use case if all the increments in the current use case have been picked and added to the system. In the same way, after adding behaviour to the system from all the use cases, the current system is the final system incorporating the behaviours from all the use cases.

Next by using X-MAN', we illustrate our incremental construction process to construct the extended bank example.
CHAPTER 5. INCREMENTAL CONSTRUCTION USING X-MAN

1. Begin Process with the first use case.
   (a) identify the initial system $S_0$ from the use case;
   (b) identify an increment $inc_0$ from the use case;
   (c) add $inc_0$ to $S_0$ to construct $S_1$;
   (d) $k:=1$;
   (e) identify an increment $inc_k$;
   (f) refactoring $S_k$ for construction if needed;
   (g) add $inc_k$ to $S_k$ to construct $S_{k+1}$;
   (h) $k:=k+1$;
   (i) repeat steps (e-h) until the last inc is identified.
2. Repeat step (i) in 1 for each remaining use case.

Figure 5.24: The construction process as pseudo code

5.5 Extended Bank Example

Now in order to illustrate a system construction using our proposed process, we consider an extended bank example. The functional requirements of the extended example are based on the simple bank system example from Figure 4.7 and the requirements of the ATM system example from [63].

5.5.1 System Use Case

In the extended bank example, the ATM system only accepts cards issued by one central bank for its many branches in a city. For simplicity, we assume two bank branches of a central bank. We write a single use case ‘withdraw money’ for the extended bank example just sufficient to illustrate our construction process.

UC1 - Withdraw Money

Brief Description. At the ATM, a card holder (client) uses his card and pin code to connect to his bank and to make a money withdraw request.

Actors. Client, Central Bank, Bank Branch.

Precondition. The Client has a valid ATM card and money in his bank account. The
ATM is operational.

*Postcondition.* The Client gets the requested amount from the cash dispenser.

*Standard Process.*

1. The client enters his ATM card in the card reader slot and then enters his pin code.

2. The client’s card data and the pin code are verified from the central bank.

3. On successful authentication, the ATM system can be linked to the client’s bank branch.

4. There are two bank branches of the central bank in the city.

5. The client enters an amount to withdraw cash which is dispensed in the cash dispenser.

6. On authentication failure, the client is given two more chances to enter pin code. After a third failure, the card is confiscated by the ATM.

7. A receipt is printed after a successful transaction.

5.5.2 *System Construction*

Now we describe incremental construction for the extended bank system specified in the use case UC1. We assume that all components with required behaviours are available in the component repository.

**Step 1, Step 2, and Step 3**

In order to use an ATM, the client enters his ATM card into the card reader slot. The behaviour to read the card is provided by the cardReader (CR) component from the component repository; so we select CR as the initial system ($S_0$), as shown in Figure 5.25.

Next, the client enters the pin code for the inserted ATM card. A component pin-Reader (PR) with a service to read the pin code is selected. The behaviour of reading card and then reading pin code is achievable by a sequencer connector SEQ1. PR
and SEQ1 (inc₀ in Figure 5.25) are added to the current system (S₀), as described in Section 5.2.1. The incremented system is S₁ in Figure 5.25.

For request authentication, the centralBank (CB) component with a service (to accept card number and pin code, and to return a boolean value) is selected from the repository. As data from S₁ is required to be passed to CB, a pipe connector PIPE₁ is also selected. CB and PIPE₁ (inc₁ in Figure 5.25) are added to the current system (S₁), as described in Section 5.2.1. In Figure 5.25, the incremented system is represented by S₂.

**Step 4 and Step 5**

CB receives a client’s card details to issue an authorisation signal. The client’s request can be sent to his bank if the card details are verified by CB. The increment to the current system is a bank (B₁) component with a service to accept account number and amount to withdraw; the output of this service is the withdrawn amount. The identified increment inc₂ (Figure 5.26) is then added to S₂ (Figure 5.25) by increasing the arity of a composition connector, as described in Section 5.2.2. The incremented system is represented by S₃. After adding the increment, by applying Rule 2 (described in Section 5.2.1), the compound service name in the pipe’s constraint is modified. Next, a new data mapping for B₁ is appended in the pipe’s constraint by using Rule 5, as described in Section 5.2.2. Changes to the PIPE’s constraint are shown in bold.

As the link establishment to the client’s bank is subject to the authorisation signal
from CB, a guard connector G1 (inc\textsubscript{3} in Figure 5.26) is added to the bank component B1 (in \(S\textsubscript{3}\)) by adding an adaptor connector, as described in Section 5.2.3. The incremented system is represented by \(S\textsubscript{4}\). After adding this increment, in the pipe’s constraint, a sub-service’s component name is changed (from B1 to G1) by using Rule 2, as described in Section 5.2.1. Next, the pipe’s constraint is extended to add data flow expression for G1 by using Rule 5, as described in Section 5.2.2.

**Step 6 and Step 7**

The ATM system serves clients of two bank branches under the organisation of the central bank. A bank (B2) component is added to a sub-component by using a selector connector SEL1 (inc\textsubscript{4} in Figure 5.27), as described in Section 5.2.1. The incremented system is represented by \(S\textsubscript{5}\). Based on the client’s request details, SEL1 forwards the request to one of the bank components in the system.

The client can enter an amount in the ATM system to withdraw cash. A read-Request (RR) component with a service (to accept an amount for withdrawal) is selected. RR is added to \(S\textsubscript{5}\) (Figure 5.27) by increasing the arity of PIPE1 and then PIPE1’s constraint is refined (to extend the data flow expression for component G1) by using Rule 5, as described in Section 5.2.2. The incremented system is represented by \(S\textsubscript{6}\).
Step 8 and Step 9

The client can enter an amount to withdraw if the authentication from the central bank is successful. Hence, $S_6$ is incremented by adding a guard $G_2$ (inc $6$ in Figure 5.28) to check successful authentication, as described in Section 5.2.3. In the incremented system $S_7$, PIPE1’s constraint is modified (by Rule 2 from Section 5.2.1) and extended (by Rule 5 from Section 5.2.2).

The withdrawn amount from the bank is dispensed in the cash dispenser of the ATM system. We identify a cashDispenser (CD) component with a service to accept the amount to dispense and to return the dispensed cash value. CD is added to a sub-component by using PIPE2 (inc $7$ in Figure 5.28), as described in Section 5.2.1. The incremented system is represented by $S_8$; no connector refinement is needed.

Refactoring

The system allows a client to enter the pin code three times for authorisation. Such repetition is achieved by adding an adaptor (finite loop) connector. The client’s card details and the pin code are checked and authenticated by CB. On authentication failure, the client is allowed to enter his pin code two more times.

Currently in system $S_8$ (before refactoring), it is not possible to add a finite loop. In order to enable the system for addition of a loop connector, $S_8$ is refactored to replace PIPE1 with two pipe connectors (Figure 5.29) by model transformation, as described
in Section 5.3.2.

The model transformation engine is executed with input model $S_8$ and developer’s input ($I = \langle PIPE1, [[SEQ1, CB], G2, G1] \rangle$); the constraint of PIPE1 is decomposed into the constraints for PIPE3 and PIPE4, as described in Section 5.3.2. The behaviour of the input and output models is the same. In Figure 5.29, interfaces of PIPE3 and PIPE4 are shown; interfaces for all other components are unchanged.

**Step 10 and Step 11**

After refactoring the current system ($S_8$), it is possible to increment the current system by adding a finite loop connector $L1$ ($\text{inc}_8$ in Figure 5.30). After adding the adaptor, PIPE3’s constraint is modified by using Rule 2, as described in Section 5.2.1. The
incremented system is $S_9$ in which the loop connector L1 keeps transferring control to the composite of PIPE4 until either the authentication is successful or the authentication is failed three times. Once the loop terminates, it sends the latest response of the login service to PIPE3.

The client’s card is confiscated by the ATM system if the authentication from the CB component is failed after three attempts. A confiscateCard (CC) component is selected and added to the system ($inc_9$ Figure 5.30) by incrementing PIPE3’s arity, as described in Section 5.2.2. The incremented system is represented by $S_{10}$.

**Step 12 and Step 13**

In order for the ATM system to confiscate the card on three authentication failures, a guard connector is selected and added to the system ($inc_{10}$ in Figure 5.31), as described in Section 5.2.3. In the incremented system $S_{11}$, a new data mapping for G3 is appended in PIPE3’s constraint by using Rule 5, as described in Section 5.2.2.
5.5. EXTENDED BANK EXAMPLE

To print a receipt, a printer (PT) component (with one service to accept card number and dispensed amount) is selected; PT is added to $S_{11}$ by increasing PIPE3’s arity ($inc_{11}$ in Figure 5.31) and a new data mapping for PT is appended in PIPE3’s constraint by using Rule 5, as described in Section 5.2.2.

**Step 14 and Step 15**

As a receipt is printed for a successful transaction, a guard connector ($inc_{12}$ in Figure 5.32) is added to $S_{12}$ to check authentication. After adding the guard G4, PIPE3’s constraint is modified by using Rule 2, as described in Section 5.2.1. Next, in PIPE3’s constraint, data flow expression for G4 in the constraint is added by using Rule 5, as described in Section 5.2.2. The incremented system $S_{13}$ is not shown separately in Figure 5.32. Finally, we add an infinite loop on top of the root PIPE connector in
CHAPTER 5. INCREMENTAL CONSTRUCTION USING X-MAN

5.5.3 Summary and Discussion

Using X-MAN’, we have illustrated the incremental construction process proposed in this thesis by applying it to the bank system. In this example, we have shown the construction process of identifying and adding an increment from the system’s use case in 15 steps. In these construction steps, we have used all defined ways of adding an
increment to the current system, as described in Section 5.2. During the construction, we have faced a situation where adding an increment to the current system was not possible without refactoring the system, as shown in Figure 5.29. The bank system is constructed and simulated in 16 steps. Using ECF, the final bank system and the output of its simulation is shown in Figure 5.33.

**Figure 5.32: $S_{13}$**

<table>
<thead>
<tr>
<th>Component</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>$s$ readCard()</td>
</tr>
<tr>
<td>PR</td>
<td>$i$ readPin()</td>
</tr>
<tr>
<td>SEQ1</td>
<td>$(s,i)$ getData()</td>
</tr>
<tr>
<td>CB</td>
<td>$b$ authorise($s,i$)</td>
</tr>
<tr>
<td>PIPE4, L1</td>
<td>$(s,b)$ login()</td>
</tr>
<tr>
<td>CC</td>
<td>confCard()</td>
</tr>
<tr>
<td>G3</td>
<td>confCard($b$)</td>
</tr>
<tr>
<td>B1, B2, SEL1</td>
<td>$i$ withdraw($s,i$)</td>
</tr>
<tr>
<td>RR</td>
<td>$i$ readAmount()</td>
</tr>
<tr>
<td>G2</td>
<td>$(i)$ readAmount($b$)</td>
</tr>
<tr>
<td>CD</td>
<td>$i$ dispCash($i$)</td>
</tr>
<tr>
<td>PIPE2</td>
<td>$(s,i)$ withdraw($s,i$)</td>
</tr>
<tr>
<td>PT</td>
<td>print($s,i$)</td>
</tr>
<tr>
<td>G4</td>
<td>print($s,i,b$)</td>
</tr>
<tr>
<td>G1</td>
<td>$((i),(i))$ withdraw($s,i,b$)</td>
</tr>
<tr>
<td>PIPE3</td>
<td>$(s,i,b)$, $[i],[i],[i]$ cshWithdraw()</td>
</tr>
</tbody>
</table>

where: $s$ = string, $i$ = int and $b$ = boolean.
Figure 5.33: The bank system in ECF
5.6 Limitations of Our Approach

In our approach to incremental construction in X-MAN, the collection of number of possible ways to add an increment to a system, the collection of system refactoring and the system construction process are referred to as the construction guidelines. One significant limitation of our approach is the construction of a sequential system with a single execution thread. Another clear limitation of our approach is the human guidance which is required to construct the system.

The prototype flow constraint language (FCL) defined for exogenous connectors has a number of limitations. For example, grouping of logical variables in an expression or nesting of boolean expressions in the selector and guard constraints is not supported by FCL. In other words, a logical expression is flat and not an expression of sub-expressions. Moreover, FCL does not support any arithmetic computations which may be needed to compute a number of service input/output values as a criterion to control the flow of control/data to the connected component(s).

The selector connector composes components by using full-signature matching algorithms, as described in Section 4.5.3. The limitation of such a selector is that the connector does not compose components with similar services but with a different number or order of output parameters. For such a situation, adapted components with guard connectors can be used. Doing this will increase the number of connectors in the control layer of the system.

5.7 Chapter Summary

In this chapter, to construct a system, we identify the set of all possible increments with respect to the proposed extensions to the X-MAN component model. Moreover, in order to enable addition of more increments to a system, we also proposed some architecture refactorings. In this chapter, incorporating aforementioned ways to increment/refactor a system, we use a construction process which constructs the system iteratively, bit by bit. In order to illustrate the construction process, we used an extended bank example and showed the incremental construction in some detail.

\[\text{This set was identified as part of work carried out jointly with my colleagues Lau, Ng and Tran [62].}\]
Chapter 6

The Common Component Modelling Example

In this chapter, we demonstrate the feasibility of our approach by applying it to construct the cash desk system of the common component modelling (CoCoME) example [88], which is a kind of benchmark in CBD. In contrast with the hypothetical bank example (used in Chapter 5), CoCoME is a non-trivial example of a real system. The cash desk system is of complex nature mainly because system’s state is required to be maintained/shared among several system components. Moreover, implementing CoCoME in X-MAN’, we found the current set of connectors is not sufficient for CoCoME; hence, we propose two new connectors and further extensions to an existing connector to address some of the limitations of our approach (described in Section 5.6). Furthermore, component sharing and adaptor relocation are proposed to increment the system behaviour; in essence, the construction guidelines are extended for using X-MAN’.

6.1 The System Overview

The CoCoME example (design by four researchers Rausch, Reussner, Mirandola and Plasil) describes a trading system in a retail business with many stores. The CoCoME system architecture comprises three different sub-systems (shown in Figure 6.1): (i) a system to handle sales at a cash desk, (ii) a system to manage the trade at the store level, and (iii) a system to manage the trade at the enterprise level. The system to handle sales (point of sale (PoS)) is operated by a cashier. In each store, there are many cash desks with a PC running the PoS system. Customers bring their purchased items for payment to a cash desk.
6.1. THE SYSTEM OVERVIEW

Each store is running a system (sSys) on a store server to manage the trade. The store manager uses the sSys system to manage trading for the store. Each cash desk PC in a store is connected to the store server. The sSys on a store server contains a central inventory which is shared by all PCs in the store. All items are registered in the inventory running on the store server. The inventory system responds to the queries from the cash desks, as well as keeping transaction records from them. A store server is connected to the enterprise server at the main office.

The enterprise is comprised of many stores at different locations. Each store server reports its transactions and inventory status to the enterprise server. The enterprise server consolidates reports from the stores to manage trading at the enterprise level. The enterprise manager uses the eSys system to manage the trading of all stores, and to facilitate logistics and re-stocking for each store.

![Figure 6.1: The CoCoME system architecture](image)

The PoS sub-system for the cash desk PCs is the key part of the CoCoME example. In a retail store, there are seven peripherals (cash drawer, keyboard, VDU (visual display unit), barcode reader, card reader, printer and light display) connected with the cash desk PC, as shown in Figure 6.2. The PoS system communicates to two external...
systems: (i) the bank system, and (ii) the sSys system running on the store server.

The PoS contains a sub-system cashBox which communicates with three peripherals (cash drawer, keyboard, and VDU). The PoS system for the cash desk directly controls other peripherals. The barcode reader is used to scan item IDs one by one; a barcode can also be entered from the keyboard if the barcode reader is unable to scan the code correctly. The VDU displays a list of scanned items with their prices. In order to accept payments from a customer’s card (read by the card reader device), the PoS system communicates to the bank system. Cash and receipts of card payments are kept in the cash drawer. The printer is used to print receipts for purchased items and card payments. The light display is used to indicate the operating mode of the cash desk (normal/express checkout).

![Diagram of PoS system]

**Figure 6.2: PoS system**

### 6.2 Extending The Construction Guidelines

Constructing the cash desk system, we realised the need to extend the construction guidelines by introducing one more way to increment the system’s behaviour. Moreover, in order to support incremental construction, we identified the need to refactor the system architecture by relocating an adaptor connector. In this section, two aforementioned expansions to the construction guidelines are described.
6.2. EXTENDING THE CONSTRUCTION GUIDELINES

6.2.1 Incrementing System by Component Sharing

Component sharing is a concept that allows more than one connector to communicate with the same component. In the deployment phase, as the intent is to construct systems, composite components are not encapsulated. Encapsulated components in X-MAN′ do not support re-entrance, as described in Section 4.1.1. In order to use the component behaviours effectively without adding unnecessary redundancy in a system, component sharing is allowed if this does not create a cycle/loop for the control flow. To avoid the creation of a cycle in the control flow, a component in the deployment phase (shown in Figure 4.6) may be shared by composition connectors at the higher level (described in Section 4.1.4) than the component.

A component maintains its state by allowing access to its data through the provided services. In X-MAN′ systems, there is only one execution thread, sharing a component does not have any adverse effect on the component’s state. A component’s state persists for the execution of a system. In case a system is required to shut down (e.g. for maintenance purposes), for later system execution from the previous system’s state, the system’s state can be persisted by storing it (in a file or database, as represented by ‘Data’ in the definition of an atomic component in Section 4.1.1) within the component.

A composite’s behaviour can contain the behaviours of its composed components in two ways: (i) the composite’s services are based on the services of the composed components (services of composed components are sub-services of the composite’s services) and (ii) the composite’s services are simply a delegation link to the services of the composed components, as described in Section 3.1.3. An example of the former case would be a composite of a sequencer (Figure 4.3(a)) and pipe (Figure 4.3(b)) connectors. An example of the latter case would be a composite of a selector connector (Figure 4.3(c)). In a composite, in the run-time phase, a selector connector forwards control to one component for a service request. That means if two components with different number of services (e.g. 6 and 8) are composed by a selector connector, the composite can accept a request for each of the services of the composed components.

In the process of system construction, there are cases when we need to compose two components to create a composite component but at the same time we also need to offer the original services of the composed components as the services of the composite, as shown in Figure 3.4. For example, in the coffee dispenser example in Figure 5.13, we would like to use the dispenser composite to dispense hot water, milk and empty cup along with the coffee service. In Figure 6.3, we extend the coffee dispenser system to
offer extra services.

In the coffee dispenser system, a composite of SEQ1 produces a white sweet coffee in cup which is the original service of the coffee dispenser in Figure 5.13. The composite of SEQ2 offers a service to dispense hot water in a cup. A service to dispense milk in a cup is offered by the composite of SEQ3. The SEL1 composite offers the services of three composites (services of SEQ1, SEQ2, and SEQ3) and the service of dispensing cup alone of the atomic component C.

Components A and D in Figure 6.3 are not shared components. In order to get extra sugar, the behaviour of a system can be increased by increasing the arity of a connector to share an existing component D, as shown inc₀ in Figure 6.3.

### 6.2.2 Relocating an Adaptor

To increment the behaviour of a system, an adaptor connector can be added to a component, as described in Section 5.2.3. An adaptor’s relocation by a developer in the control layer may add more behaviour to the system; for the incremental construction, the system behaviour after the relocation of an adaptor must contain the behaviour of the system before the relocation. Hence, such a relocation of an adaptor is a way to increment the system in our approach of incremental construction. For example, the behaviour of a system is incremented by relocating a finite loop connector to a higher
level in the control layer. After relocating an adaptor, the constraints of the relocated
adaptor can be refined by following Rule 3, as described in Section 5.2.1.

In Figure 6.4(a), we show a system with two adaptor connectors, G1 and L1. Adap-
tor G1 is a guard connector and adaptor L1 is a finite loop connector. For a service
request reaching the guard connector, the connector transfers the request on to the
component by checking the guard condition. Similarly, for each service request reach-
ing the loop connector, the loop connector adapting component \( D \) repeats the request
to the connected component.

![Figure 6.4: Relocating a connector](image)

Relocating adaptors in this system along the relocation lines (R1, R2, R3 and R4 shown in Figure 6.4(b)) must be decided by the developer of the system. Based on the
actual behaviour of the system with specific composition connectors (shown as P1, P2
and P3 in Figure 6.4), adaptors may be relocated and the constraints of the relocated
connectors can be refined by following Rule 3, as described in Section 5.2.1. More-
over, because of adaptor relocation, any affected connector’s constraint may be refined
by using concerned rules. For example, considering P1 and P2 as pipe connectors, re-
locating G1 along R1 requires us to modify the constraints of P1 and P2 by using Rule
2. Relocating a guard connector makes more behaviour be guarded by the connector.
Similarly, relocating a loop connector makes more behaviour be repeated.
6.3 Extending The Exogenous Connectors

In an attempt to construct the cash desk system of CoCoME, we faced situations where the existing exogenous connectors (as defined in Section 4.3) cannot be used to construct the required behaviour. In order to deal with such situations, we create two basic composition connectors and extend the finite loop connector (from Section 4.5.4), as described in this section. For EBNF and SoS of FCL constraints for these connectors, we will be using common terms from the legend shown in Figure 4.26.

6.3.1 A Special Selector

In X-MAN’, the selector exogenous connector (SEL) selects one of many composed components. For interface generation, SEL implements the full-signature matching mechanism, as defined in Figure 4.49. However, to construct a practical system, a selector may be required to compose components with similar services; such services have varying number and/or type of output parameters. In order to deal with such situations, we create a special selector connector sSEL with different service signature matching mechanism (shown in Figure 6.5); in this mechanism, list of out parameters of two service signatures (of the composed components) are not matched. This mechanism is referred to as half-signature matching mechanism. For interface generation of a composite, signature matching mechanism is the only difference between sSEL and SEL (from Figure 4.49) connectors. In a composite that uses sSEL, services are created without matching the set of output parameters. The control/data flow behaviour of sSEL and SEL (defined in Section 4.5.3) are the same.

\[
\text{sSig1} = \text{sSig2} \implies \begin{cases} 
\text{true, if } \text{sSig1.antt} = \text{sSig2.antt} \\
\text{sSig1.iList} = \text{sSig2.iList}, \\
\text{false, otherwise.}
\end{cases}
\]

Figure 6.5: A special selector (sSEL)

The refinement rules of SEL (from Section 5.2) can also be applied to this connector. An example of the use of this connector is shown in Figure 6.6(a). In a payment composite component, two components \(A\) for cash payment and \(B\) for card payment (with same services but different result set) are composed by a sSEL connector. The compound service of the composite contains a list of output parameters of all sub-services of the composed components. The composite’s service signatures help the
6.3. EXTENDING THE EXOGENOUS CONNECTORS

Instead of creating a new connector, the aforementioned objective can also be achieved by adding a property to the selector connector to create services for the composite by using either the full-signature or the half-signature matching mechanisms. A selector connector with such a property can be parameterised to instantiate the connector with one of the two matching mechanisms.

Despite its practical use, this connector exposes another limitation of FCL. In the run-time phase, the simple composite shown in Figure 6.6(a)) forwards control to one of the two components returning different result sets. During system construction, assuming the developer needs to pick one valid result set from the composite to pass to a service of another component. To achieve this, the developer needs to be able to check the valid result set dynamically (in the run-time phase). This kind of dynamic feature is not supported by FCL. In order to handle this limitation of FCL, we develop components to accept both kinds of result set and decide to do the right computation inside the component. For example, the print service of component $C$ shown in Figure 6.6(b) accepts both result sets from the sSEL1 composite. Hence, components like $C$ have limited reuse.

### 6.3.2 Chain of Responsibility Connector

The $PoS$ system of CoCoME is a typical example of an interactive system. In simple terms, an interactive system accepts inputs from a user from its input devices and returns outputs to the user on its output devices. In the cash desk system, there are many
devices connected to a PC running the PoS system. The cashier uses these devices to enter inputs to the PoS system. The PoS system simultaneously listens to more than one device for user input. The system may not allow the user to input into more than one device at a time. For example, while serving a customer at the cash desk, the PoS system allows the cashier to do one of three actions at a time: (i) to scan an item by using a barcode reader, (ii) to enter the item code by using the keyboard, and (ii) to push the end of sale button. This kind of behaviour in interactive systems demands listening to more than one device concurrently. In the system construction in [62], a selector connector is used to simulate the aforementioned requirement.

In X-MAN’, encapsulated components and connectors are passive elements. A system in X-MAN’ is a sequential system with a single thread of control. No component or connector has its own thread of execution, but each element performs its defined role in a system once the system control reaches it. In a system, the root level infinite loop connector makes the system thread circulate in the system. Some work has been done by Lau and Ntalamagkas to introduce concurrency into the X-MAN [64, 65]. In order to activate more than one component concurrently, a composition connector cobegin was introduced in [65]. This composition connector triggers all composed components to perform their execution.

To construct a sequential system to be able to simulate an interactive system with a demand to listen more than one device, we need a mechanism to listen to more than one peripheral before the system’s control thread proceeds sequentially. Such is the kind of requirement in the CoCoME example. In contrast to the cobegin connector which triggers components, listening to more than one device in parallel can be simulated by defining the chain of responsibility (CoR) design pattern as an exogenous connector without adding more than one control thread to the system.

In software engineering, a generic reusable solution to commonly occurring problems is referred to as a design pattern [43]. Design patterns are one of the most significant advances in software engineering to date. In this thesis, we identify a design pattern ‘chain of responsibility (CoR)’ which can be used to simulate the desired interactive behaviour in a system with a single thread of execution. According to its description in [43], the intent of CoR is to “avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request; chain the receiving objects and pass the request along the chain until an object handles it”. This design pattern defines a way of forwarding requests between a chain of handler objects (components) to find the object (component) which can handle the request. The
responder of the request is non-deterministic, but the sequence of forwarding requests to the responders is fixed. In [66], CoR is defined \(^1\) as a basic connector by using a sequencer with two constraints ‘C’ to specify the roles of the participants (components) as well as the relationships between the participants and ‘D’ to specify the flow of control to the composed participants. In this thesis, we define the CoR connector as a basic connector with constraints.

For system construction with interactive behaviours using hardware devices, we create a basic composition connector CoR. In a CoR composite, a component producing a desired response corresponds with the object handling the request in the CoR design pattern in [43]. Two or more components can be composed by a CoR if they have at least one service with no input parameters and with the equivalent non-empty result set. For example, component A with a service s1 (with signature \((\text{int, double})s1()\)) and component B with a service s2 (with signature \((\text{int, double})s2()\)) can be composed by a CoR connector.

In this section, EBNF for CoR constraints are defined/explained. Next, for a CoR connector, semantics of generation of provided interface and flows of control/data are defined. Operational semantics of CoR constraints are part of the control/data flow semantics of the connector.

**EBNF for CoR Constraints**

The EBNF grammar rule of FCL constraints for a CoR connector is shown in Figure 6.7. The syntax of CoR constraint is the same as with the guard connector; however, the semantics are different as ‘param...’ tokens (e.g. param0) in CoR’s constraint represent output parameters. For the CoR and guard connectors, the EBNF rules for the non-terminal \(\langle\text{condition}\rangle\) (shown in Figure 4.37) are same; hence, grammar rules for \(\langle\text{condition}\rangle\) are not shown in Figure 6.7.

![Figure 6.7: EBNF of FCL constraints for CoR](image)

\(^1\)A joint effort with three other colleagues Lau, Ntalamagkas and Tran.
For a service generated by a CoR connector, an FCL constraint defines the criteria (based on service output parameters) whether to return the control or to pass it to the next composed component.

A parser function (Figure 6.8(a)) in the CoR connector parses a constraint to a tuple (representing the abstract syntax tree for the constraint) of composite service ID ($sID$) and a list ($conds$) of conditions (or Boolean expressions) to forward the control/data to the composed components. A condition is a 5-tuple (shown in Figure 6.8(a)). Element $lOp$ represents a logical operator between two conditions; for the first condition, $lOp$ has value “null”. Element $pInd$ represents an output parameter index of a service and $pTy$ represents the data type of the output parameter at index $pInd$. Element $eOp$ represents the expression operator in a condition and $val$ represents the actual value in the condition. Tuples in list $conds$ are ordered by the value of element $pInd$. In a condition, the data type of a service output parameter ($pTy$) is equal to the data type of the value $val$ (represented by $val.type$ in Figure 6.8(a)) in the condition.

In order to illustrate the use of CoR constraint, a simple composite (shown in Figure 6.8(b)) $AB$ with a service $doTask$ is considered. For a service ($doTask$) request, $CoR1$ passes control to component $A$ unconditionally. $CoR1$ passes the control to the second component $B$ if the output (a Boolean parameter) of the associated sub-service of $A$ has value ‘false’; otherwise, the control is returned by the connector. The FCL constraint of service $doTask$ for the aforementioned behaviour is shown in Figure 6.8(c). A parser function in the connector parses this constraint as shown in Figure 6.8(c).

![Diagram](image-url)  
**Figure 6.8: Example of a CoR constraint**
6.3. EXTENDING THE EXOGENOUS CONNECTORS

For a service in a CoR composite, the execution of a service of one or both components is part of the connector’s semantics. Hence, an FCL constraint’s operational semantics is to evaluate the condition in the constraint. The evaluation of a condition with one or more Boolean expressions is a value from set $\mathbf{T}$. The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the CoR connector for control/data flows to the connected component.

The CoR connector shown in Figure 6.8(b) with a constraint shown in Figure 6.8(c) is defined as a CP-net shown in Figure 6.10. Transitions $T1', T2', T4, T5'$ and $T6/T6'$ in Figure 6.11 define the operational semantics of a CoR constraint.

Interface Generation

The interface generation semantics of a CoR connector with default connector properties (with $L = \{\}$ and $Z = \{\}$) are shown in Figure 6.9. For CoR, a composed component is taken as an input of set ($cSigs$) of provided services and output is a set ($pSigs$) of provided services for the composite.

\[
\begin{align*}
\text{pSigs} & \quad \text{Where:} \\
cSigs & = \{cSig_1, cSig_2, \ldots, cSig_k\}, \\
cSig & = <sID, sSig>, sID=i, \\
pSigs & = \{pSig_1, pSig_2, \ldots, pSig_n\}, \\
pSig & = <sID, sSig, sList>, sID=j, \\
sList & = [sID_1, sID_2], |Z| = |pSigs|, \\
sSig & = <sName, oList, iList>, \\
i, j & \in \{\text{positive integers}\}.
\end{align*}
\]

Figure 6.9: Interface generation

A service signature in $cSigs$ is a tuple ($<sID, sSig>$) of service identifier (a positive integer value represented by $sID$) and service signature ($sSig$). A service ($pSig$) of $pSigs$ is a 3-tuple of ($<sID, sSig, sList>$) service identifier ($sID$), service signature ($sSig$) and a list ($sList$) of sub-services’ IDs of the composed components in component connecting order. Service signature ($sSig$) is a 3-tuple of ($<sName, oList, iList>$) service name ($sName$), list of output data types ($oList$) and a list of input data types
For a service of CoR composite, list of sub-services $sList$ must have exactly one sub-service from a composed component.

In a CoR composite, for a matched service (with the empty list of input parameters and with the non-empty list of output parameters) from the composed components, a compound service (with the reference of matched services from the composed components) is added into the provided interface of the composite. For CoR, the rigorous process of interface generation is given in Appendix C.4.

In the deployment phase, a composition connector property for propagated services can be changed. The default property (with $L = \{\}$) propagates all services of the composite. A system designer can change this property by adding only desirable services to be propagated into the set of provided services of the composite component. An example of interface generation for a composite created by a CoR connector is shown in Figure 6.8.

**Control and Data Flow**

The control/data flow semantics of a CoR connector ($CoR1$ in Figure 6.10) are shown with the help of a CP-net inside the connector in Figure 6.10.

![Figure 6.10: Control and Data Flow of CoR1](image-url)
On receiving a request message for a service from outside the connector, the connector initiates sub-service requests (to the composed components) with data received in the request message. Control is passed to the first composed component unconditionally and then to the second component based on the FCL constraint of the connector. For Figure 6.10, subpage for Group1 is shown in Figure 6.11.

Figure 6.11: Group 1 of CoR1

Transition $T1'$ along with its associated places $Req$ and $Temp1$ (instead of $Temp$) of the CoR connector is exactly the same (with one variation that it does not write to place $mem$) as with the invocation connector. Transition $T1'$ (shown in Figure 6.12) is fired concurrently with $T1$. $T1'$ is fired to place the count of Boolean expressions in the parsed FCL constraint (from place $Const$) to place $cCount$. This transition writes the expressions to place $sConst$.

Place $cI$ (with colour for $pSig$ shown in Figure 6.9) in the CP-net maintains a list of provided services of the composite created by the connector. Transition $T2$ (shown in Figure 6.12) is enabled and fired if the requested service is found in place $cI$. List of sub-services $sList$ for the requested service is passed along with list of service arguments $aList$ to place $subSrvs$. Transition $T2'$ (shown in Figure 6.12) is fired to write a pair of the first expression (at location 0 stored in place $index$) from place $sConst$ and an increment to current index (from place $index$) of the expression
For each composed component to a CoR connector, there is a place (\(cI_1\) for \(A\) and \(cI_2\) for \(B\)) that maintains necessary details of sub-services from the composed components; a token in these places is a tuple \((sID, argCount)\) of a sub-service ID \(sID\) and total number of arguments \(argCount\) for the sub-service. Transition \(T_3\) (shown in Figure 6.13) initiates a sub-service request (by getting the service ID from place \(subSrvs\)) to the first composed component by sending a token to place \(CPo_1\). The response of the first sub-service is received by place \(CPi_1\).

There can be more than one expressions in a service constraint’s condition. The first element of an expression from place \(aCond\) can be ‘null’ (which will processed by transition \(T_4\) (shown in Figure 6.13). For simplicity, transitions for the processing of an expression with the first element ‘and’ (processed by transition \(T_{4}'\) shown in Figure 4.40) or ‘or’ (processed by transition \(T_{4}''\) shown in Figure 4.40) are not included in Figure 6.10.

Initially, place \(result\) holds an integer value 0 to represent ‘false’ for the condition for the requested service. Transition \(T_{5}'\) (shown in Figure 6.14) sends the intermediate
6.3. EXTENDING THE EXOGENOUS CONNECTORS

result (true/false) for processing a condition (by transition \( T4 \)) to place \( \text{result} \).

\( T5' \) passes the index for getting the next expression to place \( \text{index} \). All expressions in place \( s\text{Const} \) are processed in the explained way. If the final result of expressions evaluation is true, then transition \( T6 \) (shown in Figure 6.14) fires to write to place \( \text{Temp2} \); otherwise, transition \( T6' \) (shown in Figure 6.15) fires to write to place \( \text{Temp6} \).

On getting the successful result for the constraint condition from place \( \text{Temp6} \), transition \( T7 \) (shown in Figure 6.14) creates the response for the requested service. On getting the unsuccessful result for the conditions from place \( \text{Temp6} \), transition \( T5 \) (shown in Figure 6.15) creates the sub-service request to the second composed component. On getting the response from the last composed component, transition \( T7' \) (shown in Figure 6.15) creates the response for the requested service.

The description of the simulation (in the CPN tools) of the CoR connector (shown in Figure 6.10) is provided in Appendix B.9.
CHAPTER 6. THE COMMON COMPONENT MODELLING EXAMPLE

Figure 6.15: Transitions $T_5$, $T_6'$ and $T_7'$ in CoR CP-net

CoR Connector in CoCoME

From the cash desk system in CoCoME, we consider the example of reading a product code from two devices, keyboard (KB) and barcode scanner (BcR), as shown in $S_0$ in Figure 6.16. In the system, on receiving control, the CoR connector listens to the composed components in sequence and releases control on receiving an expected response.

As with the refinement of the constraint of the guard connector (in Section 5.2), a rule (for constraint refinement) is defined for CoR’s constraint refinement.

[Rule 6] Modify CoR’s constraint. After adding an increment to a sub-component of a CoR connector, the constraint may be modified to change the termination condition based on existing parameters. Moreover, the constraint may be extended by adding more parameters to be used for CoR termination. For example, a FinishSale (FS) component is added to CoR in $S_0$ by increasing the arity of the connector (described in Section 5.2.2) and the CoR’s constraint is extended. This rule can be defined as a procedure `modConst(sID,strExp)` with service ID (sID) and one parameter (strExp) to
6.3. 

6.3.1 Extending the Exogenous Connectors

represent the modified constraint. This procedure overrides the existing constraint of the connector with this parameter.

As with the other constrained composition connectors, CoR may also be replaced by more than one connector of the same type, as described in Section 5.3.2.

An atomic (computational or interactive) component in X-MAN' returns a response for every request, as described in Section 4.1.1. Some atomic components (e.g. PT and CC) used in the bank example (from Section 5.5) do not return a response with any data value(s). Components compatible for CoR must return a response with some data value(s); based on these data values, CoR decides to return control to the system. The required service signature in a component’s interface may be checked before a component is selected to be composed by a CoR connector. Any extra information about a service and its parameters can be attached as annotations, as described in Section 4.2.2. For smooth execution of functional computations, we assume that components in a CoR composite cannot block the control flow for a longer time span. Using ECF (from Section 4.7) a developer can view a component’s interface and annotations for a service and its parameters. However, the tool does not yet provide any features to search a component based on its service annotations automatically.

6.3.3 Extending The Finite Loop Connector

In Section 6.2.2, incrementing the system behaviour without adding any new component/connector was proposed and described. Here, the motivation for extending the finite loop connector is described with the help of the payment system in the CoCoME cash desk system ($S_0$ in Figure 6.17). Component A accepts the payment mode from the cashier (‘0’ for cash and ‘1’ for card payments), component B processes the cash payment and card payment is processed by component C.

The first time payment mode (to pay by cash or by card) selection by the cashier is not shown as part of $S_0$ (Figure 6.17 (a)). System $S_0$ has a loop connector to accept payment by card; the loop iterates ‘n’ (some number, e.g. 5) times with a break condition for successful payment. As the system cannot proceed until the card is authorised for payment, in the card payment mode, the cashier is required to change the payment mode if needed. This may be achieved by incrementing the system and relocating the loop connector along IR0, as shown in $S_1$ (Figure 6.17 (b)).

Relocating the finite loop connector may be required to deal with a situation where the values for the input arguments for the adapted component’s service (under iterative execution) are modified fully/partly. For example, relocating the loop connector along
CHAPTER 6. THE COMMON COMPONENT MODELLING EXAMPLE

IR0 in $S_1$ can only fulfill the required functionality if the next iteration (after the switch of the payment mode) executes $S_1$ with the new payment mode. Hence, to modify the values of input parameters of a requested service, mappings between output parameters (of the service) to the input parameter (of the service) are required. FCL (in Section 4.2.3) as well as the behavior (in Section 4.5.4) for the existing finite loop connector do not support these features.

**EBNF for Extended FLoop Constraints**

In order to support the aforementioned mapping features, EBNF grammar for the finite loop (from Figure 4.57) can be extended as shown in Figure 6.18.

Figure 6.18: Extended EBNF grammar for the finite loop connector

In the extended constraint for a service in the finite loop connector, in a mapping expression, a `param` token on the left hand side of `=` symbol represents an input
6.3. EXTENDING THE EXOGENOUS CONNECTORS

parameter of the service and a `param` token on the right hand side of `=` symbol represents an output parameter of the service. System $S_2$ with the extended finite loop is shown in Figure 6.19.

![Figure 6.19: Payment system $S_2$ in CoCoME](image)

In the card payment mode (in $S_2$), if the payment mode is changed, the mapping part of the constraint will change the input parameter from card to cash payment. Hence, sSEL1 is able to select component $B$ for cash payment. Initially the system control is transferred to the loop connector with two input arguments for payable amount and payment mode. The system control is transferred to $B$ if the second argument is ‘0’. Control is transferred to PIPE1 if the payment mode is ‘1’. In this mode, the system asks to change the mode by transferring control to $A$; this is a redundant action in the first iteration. The system control is returned to the root loop connector with a failure result if the card payment is failed. The loop repeats until the payment is successful within the loop’s bound.

The parser function and a simple example from Figure 4.58 is extended for the floop extension as shown in Figure 6.20. The parsed constraint tuple (Figure 6.20(a)) is extended to have a list (`maps`) of pairs (`⟨ip, op⟩`) for parameter mappings. In a pair, `ip` represents an input parameter index and `op` represents an output parameter index of a service. The data type of the input parameter (represented by `ip.type`) is equal to the data type of the output parameter (represented by `op.type`) in a pair. Other elements (described in Section 4.5.4) of the parsed constraint remain unchanged.

In order to illustrate the use of extended floop constraint, a simple adapted component $A'$ from Figure 4.58(b) is modified. As shown in Figure 6.20(b), service `func` of
A is extended with more input parameter. For the first iteration, $FL1$ forwards the request to the connected component unconditionally. Then $FL1$ iterates (after assigning the values of output parameters to the input parameters as given in the parameter mapping expressions in the constraint) the request to the connected component based on an output parameter value of the requested service. Hence, for service $\text{func}$ of component $A'$, one of the two constraints (shown as input to parser function in Figure 6.20(c)) is added to the connector. A parser function in the connector parses this constraint. In Figure 6.20(c), the first constraint ($\text{func}\{\text{param0=true or iterate 3 times withmap param1=param0}\}$) means the iteration of service $\text{func}$ until the first output parameter is not true within total 3 iterations. Second constraint ($\text{func}\{\text{iterate 3 times withmap param1=param0}\}$) means the iteration of service $\text{func}$ for three times.

The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the floop connector for control/data flows to the connected component.

For a service in a floop adapted component, the execution of a service is part of the connector’s semantics. Hence, an FCL constraint’s operational semantics is to evaluate the condition in the constraint. The evaluation of a condition with one or more Boolean expressions is a value from set $T$. The information stored in a parsed constraint (a tuple generated by the parser) will be used to define the behaviour of the floop connector for control/data flows to the connected component.

The floop connector shown in Figure 6.20(b) with the first constraint shown in Figure 6.20(c) is defined as a CP-net shown in Figure 6.21. From the extension to
the CP-net for a floop connector, transitions $T_{10}$, $T_{\text{count}}$, $T_{\text{map}}$, $T_{\text{req}}$ and $T_{\text{pair}}$ in Figure 6.23 define the operational semantics of the mapping part of a floop constraint.

**Interface Generation**

Extending the constraints for floop connector does not change the interface generation semantics of the connector. Interface generation semantics of a floop connector are given in Section 4.5.4.

**Control and Data Flow**

For the extension in the floop constraint (shown in Figure 6.20), the CP-net for the floop (shown in Figure 4.60) is extended (as shown in Figure 6.21) to support the parameter mapping functionality of the connector. For this extension, added places and transitions are shown in a dashed line enclosed region (except transition $T_{10}$ of Group 3). Group 2 in Figure 6.21 represents Group 1 of Figure 4.60 with a transition to place $\text{mapping}$.

![Figure 6.21: Control and Data Flow of FL1](image-url)
For Figure 6.21, subpage for Group 2 is shown in Figure 6.22.

![Figure 6.22: Group 2 of FL1](image)

For Figure 6.21, subpage for Group 3 is shown in Figure 6.23.

![Figure 6.23: Group 3 of FL1](image)
Here, we will explain the extended net only. The extended CP-net maps the output parameters to the input parameters for all iterations other than the first iteration if the condition to iterate is satisfied. Transition $T_4$ sends the result data of a service execution to place $Temp_7$. Transition $T_5$ has a new arc and that is to send the maps list (from place $Const$) to place $mapping$. Transition $T_{pair}$ reads a pair from the maps list from $mapping$ and sends the pair to place $aPair$. Transitions $T_5$ and $T_{pair}$ are shown in Figure 6.24.

Figure 6.24: Transitions $T_7$ and $T_8$ in the extended Floop CP-net

For a pair $(l_1, l_2)$ from place $aPair$, transition $T_{10}$ (shown in Figure 6.25) splits the list of input arguments (read from $Temp_2$) into three lists to places $L_1$, $L_2$ and $L_3$.

Figure 6.25: Transition $T_{10}$ in the extended Floop CP-net

$L_1$ is passed a list of input arguments from the beginning to the element at location $l_1 - 1$. $L_2$ is passed a list of one value from location $l_2$ from the list of result data
CHAPTER 6. THE COMMON COMPONENT MODELLING EXAMPLE

from place Temp7. L3 is passed a list of input arguments from the element at location l1 + 1 to the end of the list. Transition Tcount (shown in Figure 6.25) is enabled when the default value in place pCount is 0. This transition write the length of maps from place mapping to place pCount.

Transition Tmap creates the service request by merging the three lists from places L1, L2 and L3. CP-net with transitions Tpair, T10, and Tmap repeats for each pair (representing a parameter mapping expression in the service constraint) in the maps list from place mapping. After executing all pairs, transition Treq moves the service request with updated/modified list of arguments from place Temp2 to place CPo. The working of the rest of the CP-net was explained in Section 4.5.4. Transitions Tmap and Treq are shown in Figure 6.26.

Figure 6.26: Transitions Tmap and Treq in the extended Floop CP-net

The description of the simulation (in the CPN tools) of the flop connector (shown in Figure 6.21) is provided in Appendix B.10.

6.4 Cash Desk System

The functional requirements for the CoCoME example are described in 8 use cases in [88]. Use cases 1 and 2 describe the functional requirements for the PoS system for a cash desk. Use cases 3, 4, 5 and 7 describe the functional requirements for the sSys for the store server. The enterprise server’s functional requirements are described in use cases 6 and 8.

Next, following the incremental construction approach proposed in this thesis, we write use cases UC1 and UC2 (from [88]). The PoS system is constructed in 26 steps
6.4. CASH DESK SYSTEM

from UC1 and UC2.

For the cash desk system, all system diagrams (without component interface table and connector constraints) are shown in Appendix D.

6.4.1 Use Case 1 (UC1)

*Brief Description.* This use case is about the sale process conducted at each cash desk. In this process, customers check out by paying for the purchased items to a cashier at the cash desk.


*Precondition.* The cashier is ready to serve the next customer coming to the cash desk with the purchased items.

*Postcondition.* The customer has paid for purchased items and received the receipts. The sale is registered with the store inventory.

*Standard Process.* The standard process for UC1 is comprised of the following functional activities.

1. On arrival of a customer at the cash desk, the cashier starts the new sale process by pushing the ‘New Sale’ button on the cashBox.

2. The cashier starts entering the item identifier (ID) to the system by using either the keyboard on the cashBox or the barcode scanner.

3. The entered item details (e.g. description, price, etc.) are retrieved from the store inventory and displayed on the VDU of the cashBox with a running total of the scanned items.

4. The cashier presses the ‘Sale Finished’ button after entering all purchased items into the system.

5. Next, the cashier selects either cash payment (by pressing ‘Cash Payment’ button) or card payment (by pressing ‘Card Payment’ button) mode.
6. In the cash payment mode, the cashier enters the amount received; the system displays the entered money and the change due on the VDU. The system also opens the cash drawer. After returning the change amount, the cashier closes the cash drawer.

7. In the card payment mode, a customer enters a bank card in the card reader and enters the pin code. The card details and the pin code is sent to the bank for validation. If the card is declined, the customer tries again until the card is accepted. The cashier may switch the payment mode to cash payment at any time.

8. After successful payment, the payment transaction is recorded and a receipt is printed on the printer.

Incremental Construction

We construct the system incrementally from UC1 in 17 steps by following the incremental construction process, as described in Section 5.4. To avoid cluttering, in each figure shown in this section, component interfaces and constraints are shown for the latest incremented system.

Step 1, Step 2, Step 3, Step 4 and Step 5

The standard process of UC1 describes the sale process. The cashier presses the ‘New Sale’ button when a customer arrives at the cash desk with a shopping cart. Component startSale (SS) is selected as initial system \( S_0 \), as shown in Figure 6.27(a).

The cashier can enter an item’s code into the system by using a keyboard. Using a sequencer connector SEQ1, a keyBoard (KB) component is added to \( S_0 \) (inc\(_0\)) in Figure 6.27(b)), as described in Section 5.2.1. The incremented system is represented by \( S_1 \).

Alternatively, the cashier can use a barcode reader to scan an item’s code. A barcodeReader (BcR) component is added to sub-component KB in \( S_1 \) by using a CoR connector (inc\(_1\) in Figure 6.27(c)), as described in Section 5.2.1. The incremented system is represented by \( S_2 \).

An entered item’s details (e.g. description, price etc.) are retrieved from the store inventory and displayed on the VDU connected with the cash box. The running total of all entered items is also shown on the VDU. From the repository of components,
we identify three components SI (storeInventory), SL (shoppingList) and Disp (display) for the next increment. Component SI represents store inventory with services ‘getPrice’ to get an item’s price and ‘update’ to update the inventory. Component SL provides a service ‘addItem’ to accept an item code and price; this service returns a list of all items and the total price. SL also provides a service to clear the list. Component Disp provides services to print data on the VDU and to clear VDU. As the entered ID of an item is first retrieved from the store inventory and then added to a list of purchased items and then displayed on the VDU, we identify the behaviour of pipe connector which allows passing data between the composed components. The next increment \((inc_2)\) for the current system is the aforementioned three components and
the pipe connector. This increment is added to a sub-component (Figure 6.27(d)) in the current system by composition, as described in Section 5.2.1.

Next, components SL and Disp are shared with the SEQ1 connector and this way of incrementing the system \(inc_3\) is referred to as incrementing by increasing connector arity (in Section 5.2.2) and as component sharing (in Section 6.2.1). With this increment, on start of a new sale, the SL and Disp screen components are cleared.

Components SI, SL and Disp have more than one service. Attribute \(L\) (defined in Section 4.3) of PIPE1 has one selected service with four sub-services ‘getCode’, ‘getPrice’ (of SI), ‘addItem’ (of SL) and ‘display’ (of Disp). Similarly, attribute \(L\) of SEQ1 has one selected service with four sub-services ‘newSale’, ‘prosItem’, ‘clsList’ (of SL) and ‘clear’ (of Disp).

**Step 6, Step 7, Step 8, and Step 9**

As the sale ends by pressing the finish sale button, we identify FinishSale (FS) component that provides the behaviour to read a button click for end of sale. FS is added to the current system by increasing the arity of the CoR connector (increment \(inc_4\), Figure 6.28(a)), as described in Section 5.2.2.

After starting a new sale, the system listens to three devices (the keyboard of cash box, the barcode reader, and the finish sale button) in sequence and the cashier can only interact with one device at a time. Constraint of CoR1 is modified by using Rule 6, as described in Section 6.3.2.

After the start of new sale, the cashier interacts with the system repeatedly to enter purchased item codes until the last item is processed. Then the cashier clicks the finish sale button to proceed towards payment activity of the sale process. In the system, to achieve the repetition behaviour, a loop connector is added in the system (\(inc_5\) in Figure 6.28(b)). This is a finite loop connector (with a high number value, e.g. 1000) which terminates if the finish sale button is pressed (detected by the FS component in the system). This way of incrementing a system is defined as incrementing by adding an adaptor, as described in Section 5.2.3.

By pressing the finish sale button, the system allows the cashier to select a payment mode. A paymentMode (PM) component with a service (to read the cashier’s selection for payment) is added to a sub-component in the current system by composition (\(inc_6\) in Figure 6.28(b)), as described in Section 5.2.1.

For processing the cash payment, a cashPayment (CP) component is added with the current system by incrementing the arity of PIPE2, as described in Section 5.2.2. CP in
6.4. CASH DESK SYSTEM

inc_7 (Figure 6.28(b)) provides a service for accepting a cash amount and to calculate the change for return.

Step 10, Step 11, Step 12 and Step 13

After calculating the change amount using CP, the system displays the amount given by the customer and the change amount on the VDU. The system is incremented by adding Disp to sub-component CP (inc_8 in Figure 6.29), as described in Section 5.2.1; Disp is shared, as described in Section 6.2.1. In this increment, a pipe is added to compose a sub-component CP of the current system with another sub-component Disp in the current system.

In the cash payment mode, the cashier enters the cash amount given by the client in the system through keyboard. Clicking the enter button, the cash amount given by the customer and the remaining change are displayed on VDU, and then the cash drawer is opened. The cashier closes the drawer after returning the remaining change. A cashDrawer (CD) component is identified that offers a service to open the cash drawer
of the cash box. The identified component is added to the system by incrementing the arity of PIPE3 (inc9 in Figure 6.29), as described in Section 5.2.2. Attribute L2 of PIPE3 has one selected service with three sub-services ‘getCash’, ‘display’ (of Disp) and ‘open’.

As the cashier can select either the cash payment mode or the card payment mode, a new increment to a sub-component in the system (inc10 in Figure 6.29) is added, as described in Section 5.2.1. This increment handles the card payment. The selector connector in the increment checks the payment mode selected by the cashier (read by component PM) and transfers control either to cash payment component (PIPE3) or to PIPE4 (the composite of CR and Bnk). Component CR reads a card and pin code from the card reader device. PIPE4 passes the card details and the pin code to the Bnk.

---

2This should not be confused with the loop connectors, e.g. L1.
component for authentication (and payment). The Bnk component checks the validity of customer’s card and then transfers the bill amount from the linked account to the store account. After adding the increment, PIPE2’s constraint is modified by using Rule 2 (as described in Section 5.2.1) and extended by using Rule 5 (as described in Section 5.2.2).

In the card payment mode, if the card validation from the Bnk component is failed, the system allows the customer to re-enter the pin code until card is accepted. Repetition of card validation is achieved by adding a finite loop (to iterate 5 times) connector \((inc_{11}\) in Figure 6.29), as described in Section 5.2.3. Constraint of sSEL1 is modified by using Rule 1, as described in Section 5.2.1.

**Step 14, Step 15, Step 16 and Step 17**

To change the payment mode (described in Section 6.3.3), by sharing PM and by adding an increment to a sub-component (described in Section 5.2.1), \(inc_{12}\) (Figure 6.30) is added to the system. Then sSEL1’s constraint is modified by using Rule 1, as described in Section 5.2.2. Next, L2 is relocated along IR0 (described in Section 6.2.2) then L2’s constraint is modified by using Rule 3 and PIPE2’s constraint is modified by using Rule 2, as described in Section 5.2.1.

After payment, in order to update the store inventory, the system is incremented by increasing the arity of PIPE2 (as described in Section 5.2.2) to include SI’s update service by component sharing (as described in Section 6.2.1). This is shown as \(inc_{13}\) in Figure 6.31.

Having added \(inc_{13}\), PIPE2 passes the control flow to SI to update the sale. PIPE2’s data mapping is appended to update the store inventory by using Rule 5, as described in Section 5.2.2.

To print receipts of purchased items on a printer, component Prt is added to the system by incrementing the arity of PIPE2 \((inc_{14}\) in Figure 6.31), as described in Section 5.2.2. Next, PIPE2’s constraint is refined to print a receipt by using Rule 5. Attribute \(L\) of PIPE2 has one selected service with a sub-service ‘update’ from SI. With this increment, the sale process for UC1 is constructed incrementally; the system in Figure 6.31, is the complete system architecture for UC1 in X-MAN'.
6.4.2 Use Case 2 (UC2)

Brief Description. Use case 2 specifies the express checkout mode which is an extension to the normal checkout mode specified in UC1. Based on some condition (not specified in the original example) the system switches to the express checkout mode; we assume that this condition is based on the transaction history from the store inventory (SI). We also assume that the system determines the checkout mode after each sale transaction.

Based on the transaction history, the PoS system switches to the express checkout mode automatically. The change of colour in the light display indicates that the system is running in the express checkout mode. In this mode, customers are restricted to checking out at most 8 products, and credit card payment is prohibited.


Precondition. In the end of latest sale, the cash desk is either in normal mode or in express mode.
Postcondition. The cash desk has been changed into express/normal mode with colour green/black in the light display.

Standard Process. The standard process for UC2 is comprised of following functional activities.

1. On meeting the switching condition, the system switches from normal to express checkout mode and the system turns the light bulb green from black.

2. Payment by card is not permitted in the express checkout mode.

3. A cashier can decide to switch back to normal checkout mode while running the system in the express mode.

4. If the cashier selects the normal mode, the colour of the light turns back to black. The card payment mode is permitted for purchasing any number of items.
Incremental Construction

We construct the system incrementally for UC2 in 10 steps (starting from the system created by step 18) following the incremental construction process as described in Section 5.4.

Step 18, Step 19, Step 20 and Step 21

Next, four increments to deal with the PoS system’s operational mode are shown in Figure 6.32; component interfaces and connector constraints for $S_{19}$ are shown in Figure 6.33.

PoS System checkout mode is changed to express mode automatically and can be changed to the normal mode manually. In express mode, the card payment button is disabled. From the repository of developed components, a Mode (Mod) component stores the system’s operational mode. Mod provides services to read the system’s operational mode, to switch the mode from normal to express and to change the mode back to normal. Mod is added to the system ($S_{15}$ from Figure 6.31) by increasing
the arity of PIPE2, as described in Section 5.2.2. After printing a receipt, the Mod component evaluates the transaction history sent by SI (a double value) and switches to the express checkout mode.

Switching to express checkout mode disables card payments. A switchCardPay-ment (sCrdPay) component is identified to disable the card payment button in the keyboard attached to the cash box. The identified component is added to a sub-component in the system by composition ($inc_{16}$), as described in Section 5.2.1. The system is incremented again by adding guard G2 ($inc_{17}$), as described in Section 5.2.3; G2 checks whether the checkout mode has changed. If the mode is changed, G2 passes control to sCrdPay to switch the payment mode.

A LightBulb (LB) component, that offers a service to change the colour of the light display, is added to a sub-component in the system by composition ($inc_{18}$), as described in Section 5.2.1. Lastly PIPE2’s data mapping is appended for PIPE6 by using Rule 5, as described in Section 5.2.2. Attribute $L$ of PIPE6 has one selected
service with a sub-service ‘switch’ from Mod.

**Step 22, Step 23 and Step 24**

In the express checkout mode, the system only allows processing up to 8 items. In order to handle such a constraint, the system is incremented to share the Mod component (described in Section 6.2.1) by incrementing the arity of PIPE2 ($inc_{19}$ in Figure 6.34) as described in Section 5.2.2. Doing this increment, after the new sale button click, the system can read the status of the checkout mode. Attribute $L$ of PIPE2 has a service with a sub-service ‘getMode’ of Mod. Component interfaces and connector constraints for $S_{22}$ are shown in Figure 6.35.

In increment $inc_{20}$ (Figure 6.34), a counter (Cnt) component to count the processed items is added in the system by increasing the arity of PIPE1, as described in Section 5.2.2; the added component is also shared with SEQ1 in the same increment, as described in Section 6.2.1. Attribute $L$ of SEQ1 has a service with a sub-service ‘reset’ of Cnt and attribute $L$ of PIPE1 has a service with a sub-service ‘inc’ of Cnt.
6.4. CASH DESK SYSTEM

In order to add the next increment in the system, we face a difficulty because of the limitation of FCL constraints for the guard connector. We assume that the FCL constraints of the guard connector can have sub-expressions in the logical expression. In Figure 6.34, increment \( \text{inc}_{21} \) adds guard G3 to the system, as described in Section 5.2.3. Logical expression of G3 is an expression of sub-expressions. G3 allows control (request for service ‘addItem’ of SL) to pass through if the system is operating in normal mode or number of processed items are less than 9 in express mode.

After adding these increments, constraints of PIPE1 and PIPE2 are refined by using Rule 2 and Rule 5, as described in Section 5.2.1 and in Section 5.2.2 respectively.
Step 25 and Step 26

The cashier can switch from the express checkout mode to the normal checkout mode. To achieve the desired behaviour, the system is incremented ($inc_{22}$ shown in Figure 6.36) by composition (as described in Section 5.2.1) and by component sharing (as described in Section 6.2.1). PIPE7 composes SEQ1 with two sub-components (Mod and SEQ2) of the system by component sharing, as described in Section 6.2.1. Next a guard is added to the system (shown as $inc_{23}$), as described in Section 5.2.3. In Figure 6.36, interface and constraint of G4 and PIPE7 are shown; the rest of the component interfaces and connector constraints are the same as in Figure 6.35.

![Diagram](image-url)
On system execution, PIPE7 transfers control to Mod to execute ‘toNMode’ (to ask the user to switch mode if the system is running in express mode); this service returns an integer value and the current operational mode (express/normal mode) of the system. The integer value ‘0’ represents no change and value ‘1’ represents change of mode. Then, PIPE7 passes control to G4 which passes control to SEQ2 to enable/disable card payment and switch the light colour if the mode was changed by the user. Next, PIPE7 passes control to SEQ1 to start a new sale process. Attribute L of PIPE7 has a sub-service ‘toNMode’ from Mod.

The cash desk system of the CoCoME example is constructed incrementally in 26 steps. In the start, the system transfers control to the Mod component. If the system is in express mode, the system asks the user to change the mode; otherwise the system continues its control flow. The system implements the sale process specified in UC1 and in UC2.

6.4.3 Summary

By carrying out the construction process for all behaviours specified in use cases UC1 and UC2, the PoS system for the cash desk is constructed in 26 incremental steps (creating systems $S_1$ to $S_{24}$). The summary of mechanisms used to add increments/behaviour is shown in Figure 6.37.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Construction Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding increments to the current system by composition</td>
<td>2,25</td>
</tr>
<tr>
<td>Adding increments to a sub-component by composition</td>
<td>3,4,8,10,12,14,19,21</td>
</tr>
<tr>
<td>Adding increments by increasing composition connector arity</td>
<td>5,6,9,11,16,17,18,22,23</td>
</tr>
<tr>
<td>Adding guards as increments</td>
<td>14,20,24,26</td>
</tr>
<tr>
<td>Adding loops as increments</td>
<td>7,13</td>
</tr>
<tr>
<td>Adding behaviour by component sharing</td>
<td>5,10,14,16,22,23,25</td>
</tr>
<tr>
<td>Relocating an adaptor</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 6.37: Mechanisms for adding increments/behaviour in the PoS system construction

The first step of construction is to select an initial system and the rest (25 steps) of the construction steps increment the behaviour of the system, as shown in Figure 6.37.
In the incremental construction of the cash desk system of CoCoME, 2 increments are added to the current system by composition.

In this example, considering the use cases for the cash desk system, the summary of mechanisms (to add increments) to construct the system in X-MAN' shows that the composition connectors play the key role in system construction.

In this section, we demonstrate the feasibility of our approach for building systems incrementally by applying it to construct the cash desk system of CoCoME. As CoCoMe example is a benchmark in CBD, we can conclude that our approach is useful in building large systems in small steps incrementally. With the help of this example, we were able to identify and define a new basic composition connector for X-MAN'. Moreover, we also identified ways to extend some exogenous connectors for their practical use. Furthermore, component sharing and adaptor relocation were also recognised as ways to add behaviour to a system. However, implementing CoCoME example, we also have learned a practical situation faced due to the limitation if FCL constraints for the guard connector.

Using our tool ECF (described in Section 4.7), we constructed and simulated the cash desk system in 26 steps. In the ECF implementation, the final cash desk system from Figure 6.36 is simplified due to the limitations of our approach (described in Section 5.6) and the ECF (described in Section 4.7). Moreover, the behaviour of a CoR connector (defined in Section 6.3.2) is simulated by using pipe and guard connectors, as shown in Figure 6.38. After each step, the system is simulated to verify the functional behaviour and the control/data flows of the system. The system is designed using ECF tool (Figure 6.38) and simulated in NetBeans as ECF does not simulate systems which require user inputs from the text terminal (e.g. command prompt).

6.5 Chapter Summary

In this chapter, in order to evaluate the feasibility of our incremental construction approach, we constructed a component-based system for the cash desk system in the CoCoME example. Using X-MAN', we constructed the cash desk system by identifying the system behaviours from the two use cases (of the CoCoME example) bit by bit. The system was completed in 26 incremental iterations. Moreover, in this chapter, the construction guidelines for X-MAN' system construction (from Chapter 5) are further extended to increment a system by component sharing and by relocating an
adaptor connector. Corresponding to the chain of responsibility design pattern, we proposed a basic exogenous connector that is helpful to simulate the interaction needed for CoCoME in X-MAN’. Furthermore, we proposed a special selector connector and extended the finite loop connector to implement practical systems.
Chapter 7

Conclusions and Future Work

Recommendations

In this thesis, motivated by refinement based system construction approaches in software engineering, we defined an incremental construction approach for building component-based systems incrementally and iteratively. Constructing a system iteratively and incrementally is a practical approach to build large and complex systems incrementally, i.e., iteratively increment an incomplete version of the system under construction until the system is complete. In this chapter, we draw conclusions and set directions for future research.

7.1 Conclusions

In this thesis, we have proposed a component-based approach to incremental construction. By incremental construction, we mean growing a system incrementally and iteratively until the system is complete. As humans can deal with a limited measure of complexity at a time, for constructing large and complex systems, incremental construction can provide a practical solution to manage the scale and complexity [62]. In each step, the existing behaviour of the current system is contained in the behaviour of the incremented system. To date there are many incremental construction approaches proposed to construct systems by stepwise program refinement. A refinement based incremental construction is a top-down design and development process to construct systems. In contrast, system construction in CBD is a bottom-up development and design process.

In CBD, systems are constructed by following a component model which defines
components and their composition mechanisms. By virtue of composition mechanisms, a component model defines the ways in which a system can be constructed from components. In component models, composition mechanisms are used to create composites as well as to construct systems. In the context of incremental construction, we have studied composition mechanisms defined in current component models. Amongst the current component models, in comparison to object-based component models, component models with explicit connectors define interaction (control/data flow) policies as connectors [26]; such connectors represent composition mechanisms for components. In these component models, components and connectors can both be reused. In truly CBD approaches, system construction is a bottom-up process [52, 29] and the increments for the system must be pre-built system-independent elements (component/connector).

In CBD, ADLs (architectural description languages) are the most widely used component models. In ADLs, composition of different types of architectural units by port connections create three general architectural styles: (i) pipe-and-filter style, (ii) client-server style and (iii) publish-subscribe style. A generic architectural unit can have required services (or ports); such a component with required services does not have fixed behaviour. In contrast, encapsulated components in X-MAN do not have any required ports. Hence, from our investigation of current component models, we selected X-MAN as the most suitable component model for achieving incremental construction. In X-MAN, encapsulated components are composed by exogenous composition connectors. Moreover, X-MAN defines adaptor connectors that can be added to a system as increments.

In this thesis, based on the notation of behaviour containment, we defined a novel incremental construction approach to construct component-based systems. In our approach, we construct systems by using our extended construction guidelines for the extended X-MAN component model (referred to as X-MAN'). Our proposed guidelines include the possible ways to add increments to a system and ways to refactor the system architecture to support incremental construction. Moreover, the guidelines also include the iterative process of incremental construction that requires designer (human) input. In achieving the goal of successful system construction, the designer’s choices are crucial in selecting the right components/connectors with the desired behaviour and adding these elements to the system to achieve the desired system behaviour. In our proposed approach, during the iterative construction process, the system architecture may be refactored to add further increments.
The advantage of incremental construction is its ability to tackle complexity, in the sense that it is easier to build a system bit by bit, rather than building the whole system in one go. We have constructed the cash desk system in the CoCoME example to demonstrate the feasibility of our construction approach. In our approach, after adding each increment the system has fixed behaviour and therefore the system can be simulated to evaluate the partial behaviour. Being able to construct system from pre-existing set of components/connectors and to execute/simulate a partial system is the strength of our approach. However, one significant limitation of our approach is the human guidance which is required to construct the system.

To facilitate system construction, we have developed a prototype software tool (exogenous composition framework) to construct systems visually and to simulate the constructed system. During incremental system construction, the X-MAN system after each increment is a complete system. Using ECF, we execute a test case to simulate the system after each increment. However, the tool does not directly support the incremental construction process.

The overall contribution of this thesis is to redefine the notion of incremental construction with an emphasis on the behaviour containment and to study current component models for achieving incremental construction. However, the main contribution is to achieve incremental construction by using our X-MAN component model. We developed a prototype tool (exogenous composition framework (ECF)) in which X-MAN semantics are implemented.

7.2 Future Work

In X-MAN, a system corresponds to a layered architecture style with two broad layers: the computation layer and the control layer. Adding more behaviours as increments makes the system grow horizontally (addition to the computation layer) and/or vertically (addition to the control layer). X-MAN is a unique component model in which components in a system have no dependency upon other components; the model defines a set of connectors for system construction. For practical system construction incrementally, X-MAN can be advanced/extended in a number of ways. In this section, in order to further support our approach of incremental construction, we highlight few ways to extend the X-MAN component model in future.
7.2. FUTURE WORK

7.2.1 Composite Connectors

In X-MAN', the structure of the control layer inevitably leads to big hierarchies of connectors. It is very useful to be able to reduce the complexity in such hierarchies wherever possible. This control structure can be simplified by replacing two or more connectors with a basic/composite connector. This topic requires a careful investigation for any such changes.

For future work, we recommend investigating meaningful compositions of basic composition connectors into composite composition connectors. For example, in the cash desk system (Figure 6.36), the use of guard connector with pipe may be investigated to create a meaningful composite composition connector. A composite connector can be stored into the repository of connectors for further reuse.

7.2.2 Composition Connectors for Deployment

Composition of pre-existing components to create new composites or systems is the essence of CBD. In X-MAN', the component life cycle is comprised of three phases: (i) design phase, (ii) deployment phase and (iii) run-time phase. In the design phase new generic components are created by using a builder tool which are deposited into the repository in source code or in binary code. The intent of the deployment phase is to create a system for a particular run-time environment (RTE). A deployment phase system is executable in a particular run-time environment. During execution all the binary components and connectors get instantiated for system execution.

In the deployment phase, like our proposed CoR and special selector connectors (described in Section 6.3), we recommend further research for more basic exogenous connectors that can be useful in the construction of practical systems.

7.2.3 Dynamic Incremental Composition

Currently, in X-MAN', there is no notion of composition in the run-time phase. In the X-MAN' component model, in a composite, a new component/composite can be composed by increasing the arity of a composition connector (described in Section 5.2.2). In future, we would like to investigate the issues related to add increments to a composite in the execution phase. This would make dynamic incremental composition possible in X-MAN'. In real world, running/executiong software systems can be extended without suspending its services. For example, a central database of a company is required to be extended without suspending its services to its clients.
7.2.4 Extending Flow Constraint Language

The prototype flow constraint language (FCL) has a number of limitations for fixing the behaviour of exogenous connectors, as described in Section 5.6 and in Section 6.3.1. In future, in order ease working with FCL, we plan to investigate and expand FCL to overcome the limitations of FCL in comparison to a realistic language. Moreover, instead of hand parsing, we also would like to implement a parser program for FCL in ECF. These changes would ease the learning and use of FCL.

7.2.5 Extending The Exogenous Composition Framework Tool

Exogenous Composition Framework (ECF) is a prototype tool that provides basic exogenous connectors for the support of system construction in X-MAN', as described in Section 4.7. However, the tool does not directly support the incremental construction process or provide much automated support for the developer to achieve incremental construction. Hence, in future, we intend to survey modern tools for system construction and investigate the ways to automate features (e.g. searching a component based on service annotations from a big repository of components, deciding to pick component from a number of compatible components etc.) in ECF that can help the developer in system construction. Moreover, further extension to exogenous connectors proposed in Section 6.3 are to be implemented in ECF. Furthermore, we intend to investigate ways to automate the applications of rules to refine connector constraints. This will help the system developers to save a considerable amount of development time.

7.2.6 Exception Handling

For system construction, X-MAN' does not describe how to handle exceptions in a system. Exception handling in X-MAN' system can be dealt with by a developer while developing components or constructing systems. For example, a component developer can write the computation methods for a basic component to raise exceptions by returning certain values for specific return parameters. This way of handling exception is not a realistic practice; hence, in order to support exception handling in X-MAN', we would like to investigate realistic approaches for exception handling.
7.2. FUTURE WORK

7.2.7 Supporting Concurrent Processing

Currently, our approach is applicable to sequential system construction in the X-MAN\ellation model. In future, we would like to apply our approach for concurrent system construction. In this regard, we intend to explore the component model with concurrency from [82].
Bibliography


[36] S. Ducasse, O. Nierstrasz, N. Schüri, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.


Appendix A

Operational Semantics for Connector Constraints

In this appendix, we define the structural operational semantics (SoS) [87] for FCL constraints for the constrained connectors. A constrained exogenous connector uses its FCL constraints to define its behaviour in the run-time phase. Hence, the operational semantics of FCL constraints are part of the control/data flow behaviour of their respective exogenous connectors. In this section, SoS of FCL constraints are limited to the evaluations of the constraints only. The context for the operational semantics of FCL constraints becomes clearer by reading the control/data flow semantics (shown as a CP-net) of the respective constrained connector in Section 4.5 and Section 6.3.

In order to avoid unnecessary repetition and to increase readability of this section, in Figure A.1, common terms for SoS of FCL constraints are shown.

Type sets associated with FCL constraints are shown in Figure A.1(a). Elements of set $P$ and $ID$ represents locations to hold a service parameter value and the index of a component/service respectively. Sets $T$, $R$, $I$, $S$ and $Num$ represent actual values. For the operational semantics of FCL constraints, we use metavariables (shown in Figure A.1(b)) to range over type sets shown in Figure A.1(a); these metavariables can be numbered (e.g. $s_0$, $s_1$, $t_0$, $t_1$). Figure A.1(c), shows the legend for operational semantics for FCL constraints. An overloaded function $x$ (with one input value) shown in Figure A.1(c) reads a location value from a constrained connector’s state. Thus $x(l_0)$ is the value store at location $l_0$ in a constrained connector’s state. Function $x$ (with three input values) reads the third input parameter value (a service parameter) in the context of the second parameter (the service index) in the context of the first parameter (the component index) from a constrained connector’s state.
Before describing SoS rules for the pipe connector, we emphasise that these rules for a constrained connector are limited to the evaluations of the constraints only. The context for the operational semantics of FCL constraints becomes clearer by reading the control/data flow semantics (shown as a CP-net in Section 4.5 and Section 6.3) of the respective constrained connector.

To define the operational semantics of pipe constraints, four SoS rules $R1$, $R2a$, $R3$ and $R4$ are defined as shown in Figure A.2. $R1$ defines the evaluation of values from the connector’s state $x$; a value is already evaluated with itself. $R2a$ defines the evaluation of a location (in the context of other locations) to its contents in the connector’s state. In this rule, $x(l1,l2,l3)$ gets the contents of location $l3$ (a parameter of a service at location $l2$ of a component at location $l1$). $R3$ defines the evaluation of a pipe constraint for a specific compound service (referred by the service name) with index value located at $l0$; this constraint has one data mapping construct with one data flow expression. The input parameter (with index value located at $l3$) of a service (with index value located at $l2$) of a composed component (with index value located at $l1$) is assigned the value of an output parameter (with index value located at $l6$) of a service (with index value located at $l5$) of a composed component (with index value located at $l4$). In short, $R3$ changes the state of the connector from $x$ to $x'$ by replacing the content of location $l3$.
with the content of location \( l6 \). Hence, \( R4 \) defines the evaluation of a pipe constraint for a specific compound service (referred by the service name) with index value located at \( l0 \); this constraint has one data mapping construct with two data flow expressions in sequence.

\[
\begin{align*}
\text{R1: To evaluate values} & \quad \langle v, x \rangle \rightarrow v \\
\text{R2: To evaluate locations} & \quad \langle l, x \rangle \rightarrow x(l) \\
\text{R2a: To evaluate locations in the context of other locations} & \quad \langle l0, l1, l2, x \rangle \rightarrow x(l0, l1, l2) \\
\text{R3: To evaluate pipe constraints with a single data flow expression} & \quad \langle l1, l2, l3, x \rangle \rightarrow x(l1, l2, l3) \\
\text{R4: To evaluate pipe constraints with data flow expressions in sequence} & \quad \langle l0\{l1, l2, l3, l4, l5, l6\}, x \rangle \rightarrow x'' 
\end{align*}
\]

Figure A.2: Operational semantics of pipe constraints

A.2 Operational Semantics of Guard Constraints

In a constraint, a Boolean expression has two operands and a binary operator. For a Boolean expression, SoS rules (\( R5-R10 \)) for 6 comparison operators (\( =, \neq, >, \geq, <, \) and \( \leq \)) and SoS rules (\( R11-R18 \)) for 8 operators (\( \text{has, } \neg\text{has, startswith, } \neg\text{startswith, endswith, } \neg\text{endswith, equalto, and } \neg\text{equalto} \)) for string type operands are shown in Figure A.3.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>To evaluate an expression with an equality operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) = c(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) and ( c(0) ) are equal, and is false otherwise)</td>
</tr>
<tr>
<td>R6</td>
<td>To evaluate an expression with an inequality operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) &lt; c(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) and ( c(0) ) are unequal, and is false otherwise)</td>
</tr>
<tr>
<td>R7</td>
<td>To evaluate an expression with a greater than operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) &gt; n(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is greater than ( n(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R8</td>
<td>To evaluate an expression with a greater than or equal to operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \geq n(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is greater than or equal to ( n(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R9</td>
<td>To evaluate an expression with a less than operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) &lt; n(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is less than ( n(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R10</td>
<td>To evaluate an expression with a less than or equal to operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \leq n(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is less than or equal to ( n(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R11</td>
<td>To evaluate an expression with a &quot;has&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ has } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( s(0) ) is a substring of ( x(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R12</td>
<td>To evaluate an expression with a &quot;!has&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !has } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( s(0) ) is not a substring of ( x(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R13</td>
<td>To evaluate an expression with a &quot;startswith&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !startswith } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) does not begin with ( s(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R14</td>
<td>To evaluate an expression with a &quot;startswith&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !startswith } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) begins with ( s(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R15</td>
<td>To evaluate an expression with an &quot;endswith&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !endswith } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) ends with ( s(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R16</td>
<td>To evaluate an expression with a &quot;endswith&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !endswith } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) does not end with ( s(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R17</td>
<td>To evaluate an expression with an &quot;equalto&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ equalto } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is equal to ( s(0) ), and is false otherwise)</td>
</tr>
<tr>
<td>R18</td>
<td>To evaluate an expression with a &quot;!equalto&quot; operator</td>
</tr>
<tr>
<td></td>
<td>( x(0) \text{ !equalto } s(0) \rightarrow t(0) ) (where ( t(0) ) is true if ( x(0) ) is unequal to ( s(0) ), and is false otherwise)</td>
</tr>
</tbody>
</table>

Figure A.3: SoS rules for Boolean expressions with comparison and string type operators
A service constraint in a guard connector is comprised of a condition of one or more Boolean expressions separated by an operator (\textit{and} or \textit{or}). For a guard condition, two SoS rules (R19-R20) with two Boolean expressions separated by an operator (\textit{and} or \textit{or}) are shown in Figure A.4. As the condition for a guard constraint is evaluated to a Boolean data type, a condition is also a Boolean expression. The SoS rule (R21) for a specific service (referred by the service name) with index value located at $l_0$ with a condition is shown in Figure A.4.

\begin{align*}
\text{R19: To evaluate an expression with an "and" operator} & \quad \frac{b \quad t_0 \quad b, x \quad t_1}{b \land b, x \quad t_2} \quad (\text{where } t_2 \text{ is true if } t_0 \text{ is true and } t_1 \text{ is true, and is false otherwise}) \\
\text{R20: To evaluate an expression with an "or" operator} & \quad \frac{b \quad t_0 \quad b, x \quad t_1}{b \lor b, x \quad t_2} \quad (\text{where } t_2 \text{ is true if } t_0 \text{ is true or } t_1 \text{ is true, and is false otherwise}) \\
\text{R21: To evaluate guard constraints} & \quad \frac{b \quad t_0}{l_0 \{b\}, x \quad t_1} \quad (\text{where } t_1 \text{ is true if } t_0 \text{ is true, and is false otherwise})
\end{align*}

Note: \textit{b} ranges over Boolean expressions.

Figure A.4: SoS rules for a guard constraint

### A.3 Operational Semantics of Selector Constraints

For selector constraints, SoS rules (R5-R20) for Boolean expressions (with 14 operators shown in Figure A.3) and for a condition with two Boolean expressions (shown in Figure A.4) are applicable. For a specific service (referred by the service name) with index value located at $l_0$ from two composed components (with index values located at $l_1$ and $l_2$) with a condition each, the SoS rule (R22) is shown in Figure A.5. \( R22 \) evaluates to the location of the first component if the condition for this component is true. \( R22 \) evaluates to the location of the second component if the condition for the first component is false and the condition for the second component is true.

### A.4 Operational Semantics of Finite Loop Constraints

For finite loop constraints, SoS rules (R5-R20) for Boolean expressions (with 14 operators shown in Figure A.3) and for a condition with two Boolean expressions (shown
R22: To evaluate selector constraints

\[
\begin{align*}
\langle b_0, x \rangle &\rightarrow t_0 \\
\langle b_1, x \rangle &\rightarrow t_1 \\
\langle l_0\{11--b_0;12--b_1\}, x \rangle &\rightarrow l_3
\end{align*}
\]

(where \( l_3 \) is equal to \( l_1 \) if \( t_0 \) is true and \( l_2 \) is equal to \( l_1 \) if \( t_0 \) is false and \( t_1 \) is true)

Note: \( b \) ranges over Boolean expressions.

Figure A.5: Operational semantics of a selector’s constraint

in Figure A.4) are applicable. For a specific service (referred by the service name) with index value located at \( l_1 \) without/with a condition, the SoS rule (\( R23/R24 \)) are shown in Figure A.6.

R23: To evaluate unconditional finite loop constraints

\[
\begin{align*}
\langle l_0, x \rangle &\rightarrow x(l_0) \\
\langle i, x \rangle &\rightarrow i
\end{align*}
\]

(where \( t_0 \) is true if the content of \( x(l_0) \) is unequal to \( i \), and is false otherwise)

R24: To evaluate conditional finite loop constraints

\[
\begin{align*}
\langle b_0, x \rangle &\rightarrow t_0 \\
\langle l_0, x \rangle &\rightarrow x(l_0) \\
\langle i, x \rangle &\rightarrow i
\end{align*}
\]

(where \( t_1 \) is true if \( t_0 \) is true or the content of \( x(l_0) \) is unequal to \( i \), and is false otherwise)

Note: \( b \) ranges over Boolean expressions.

Figure A.6: Operational semantics of floop constraints

\( R23 \) (rule for the unconditioned FCL constraint for the floop connector) evaluates to true if the content at location \( l_0 \) (managed by the connector and not part of the constraint’s syntax) is equal to value \( i \) (given in the constraint’s syntax). \( R24 \) (rule for the conditioned FCL constraint for the floop connector) evaluates to true if the condition \( b_0 \) evaluates to true or the content at location \( l_0 \) (managed by the connector and not part of the constraint’s syntax) is equal to value \( i \) (given in the constraint’s syntax).

A.5 Operational Semantics of CoR Constraints

The EBNF grammar rules of FCL constraints for a CoR connector is shown in Figure 6.7. The syntax of CoR constraint is the same as with the guard connector; however, the semantics are different as ‘param...’ tokens (e.g. param0) in CoR’s constraint represent output parameters. For the CoR and guard connectors, the EBNF rules for
A.6. OPERATIONAL SEMANTICS OF THE EXTENDED FINITE LOOP CONSTRAINTS

For finite loop constraints with parameter mapping, SoS rules (R25-R28) for unconditional constraints and SoS rules (R29-R30) for conditional constraints are shown in Figure A.7.

Rule R25 changes a memory location $l0$ with the value of another memory location $l1$. R26 changes two memory locations $l0$ and $l2$ with the values of other memory locations $l1$ and $l2$ in sequence.

An unconditional constraint with one parameter mapping expression, rule R27 has two cases. Rule R27(a) evaluates to true if the content at location $l0$ (managed by the connector and not part of the constraint’s syntax) is not equal to value $i$ (given in the constraint’s syntax). Furthermore, this rule also evaluates the parameter mapping expression. Rule R27(b) evaluates to false if the content at location $l0$ is equal to value $i$; this rule does not evaluate the parameter mapping expression.

Similarly, an unconditional constraint with parameter mapping expressions in sequence, rule R28 has two cases. Rule R28(a) evaluates to true if the content at location $l0$ (managed by the connector and not part of the constraint’s syntax) is not equal to value $i$ (given in the constraint’s syntax). Furthermore, this rule also evaluates the parameter mapping expressions in sequence. Rule R28(b) evaluates to false if the content at location $l0$ is equal to value $i$; this rule does not evaluate the parameter mapping expressions.

A conditional constraint with one parameter mapping expression, rule R29 has two cases. Rule R29(a) evaluates to true if the condition $b0$ evaluates to false or the content at location $l0$ (managed by the connector and not part of the constraint’s syntax) is not equal to value $i$ (given in the constraint’s syntax). Furthermore, this rule also evaluates the parameter mapping expression. Rule R29(b) evaluates to false if the condition $b0$ evaluates to true or the content at location $l0$ is equal to value $i$; this rule does not evaluate the parameter mapping expressions.
### R25: To evaluate one parameter mapping expression in finite loop constraints

\[
\langle l_0, x \rangle \longrightarrow x(l_0) \quad \langle l_1, x \rangle \longrightarrow x(l_1) \quad (\text{replace the value of } x(l_0) \text{ with the value of } x(l_1))
\]

\[
\langle l_0=l_1, x \rangle \longrightarrow x'
\]

### R26: To evaluate parameter mapping expressions in sequence in finite loop constraints

\[
\langle l_0=l_1, x \rangle \longrightarrow x' \quad \langle l_2=l_3, x \rangle \longrightarrow x'' \quad (\text{replace the value of } x(l_0) \text{ with the value of } x(l_1) \text{ then replace the value of } x(l_2) \text{ with the value of } x(l_3))
\]

### R27: To evaluate unconditional finite loop constraints with one parameter mapping expression

(a) \[
\langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is true because the value of } x(l_0) \text{ is equal to } i; \text{ hence, the parameter mapping expression is evaluated})
\]

(b) \[
\langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is false because the value of } x(l_0) \text{ is equal to } i; \text{ hence, the parameter mapping expression is not evaluated})
\]

### R28: To evaluate unconditional finite loop constraints with parameter mapping expressions in sequence

(a) \[
\langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is true because the value of } x(l_0) \text{ is unequal to } i; \text{ hence, parameter mapping expressions are evaluated})
\]

(b) \[
\langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is false because the value of } x(l_0) \text{ is equal to } i; \text{ hence, parameter mapping expressions are not evaluated})
\]

### R29: To evaluate conditional finite loop constraints with one parameter mapping expression

(a) \[
\langle b_0, x \rangle \longrightarrow t_0 \quad \langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is true because } t_0 \text{ is false or the value of } x(l_0) \text{ is unequal to } i; \text{ hence, the parameter mapping expression is evaluated})
\]

(b) \[
\langle b_0, x \rangle \longrightarrow t_0 \quad \langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is false because } t_0 \text{ is true or the value of } x(l_0) \text{ is equal to } i; \text{ hence, the parameter mapping expression is not evaluated})
\]

### R30: To evaluate conditional finite loop constraints with parameter mapping expressions in sequence

(a) \[
\langle b_0, x \rangle \longrightarrow t_0 \quad \langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_2=l_3, l_4=l_5, x \rangle \longrightarrow x'' \quad (\text{where } t_1 \text{ is true because } t_0 \text{ is false or the value of } x(l_0) \text{ is unequal to } i; \text{ hence, parameter mapping expressions are evaluated})
\]

(b) \[
\langle b_0, x \rangle \longrightarrow t_0 \quad \langle l_0, x \rangle \longrightarrow x(l_0) \quad i \quad \langle l_1, x \rangle \longrightarrow x' \quad (\text{where } t_1 \text{ is false because } t_0 \text{ is true or the value of } x(l_0) \text{ is equal to } i; \text{ hence, parameter mapping expressions are not evaluated})
\]

### Note:

b ranges over Boolean expressions.

Figure A.7: Operational semantics of floop constraints with parameter mapping
Similarly, a conditional constraint with parameter mapping expressions in sequence, rule \textit{R30} has two cases. Rule \textit{R30(a)} evaluates to true if the condition \textit{b0} evaluates to false or the content at location \textit{l0} (managed by the connector and not part of the constraint’s syntax) is not equal to value \textit{i} (given in the constraint's syntax). Furthermore, this rule also evaluates the parameter mapping expressions. Rule \textit{R30(b)} evaluates to false if the condition \textit{b0} evaluates to true or the content at location \textit{l0} is equal to value \textit{i}; this rule does not evaluate the parameter mapping expressions.
Appendix B

Coloured Petri Nets

B.1 Introduction

Coloured Petri net (CP-net) is a graphical-oriented language [54] which can be used to design, specify, simulate and verify a system. Formally, at an abstract level, CP-net is a tuple \((NS, TV, NI)\) of net structure \((NS)\), types and variables \((TV)\), and net inscriptions \((NI)\). At the detail level, CP-net is a nine-tuple \((P, T, A, \sum, V, C, G, E, I)\).

Set of places \((P)\), set of transitions \((T)\) and set of arcs \((A)\) represent net structure \((NS)\). Set of colour sets \((\sum)\) and set of variables \((V)\) represent types and variables \((TV)\) of the net respectively. Functions \(C\), \(G\), \(E\) and \(I\) represent net inscriptions \((NI)\) where:

(i) \(C\) assigns colour sets to places,
(ii) \(G\) assigns guards to transitions,
(iii) \(E\) assigns expressions to arcs, and
(iv) \(I\) assigns initial markings to places. In this section, CP-net primitives are described briefly with the help of an example modelled in the CPN tools, as CP-net primitives are used to define CT-net.

In the example CP-net shown in Figure B.1(a), there are two places \((A\) and \(B\) with colour \(INT)\), one transition \((T\) with a guard) and two arcs (arrowed lines with expressions) between the two places and one transition. Initial and current (shown in a box) markings of \(A\) are shown. Transition \(T\) is enabled as there are tokens in the input place \((A)\) to fulfil \(T\)’s guard. On firing the enabled transition, two tokens from place \(A\) are moved to place \(B\), as shown in Figure B.1(b).

In a net, a place represents a memory location to store tokens (data). Initial marking of a place (shown as a label next to the place) represents number of tokens and their values; this is required to simulate the net. In the net, tokens and their values shown in a box (next to a place) represent the current marking of the place. In the place marking, number of tokens and their value is separated by a backward single quote.
B.2. INVOCATION CONNECTOR

In this section, to simulate the control/data flows of the invocation connector in component A shown in Figure 4.19 and Figure 4.20, tokens are added in place $cI$ to represent the provided services by component A as shown in Figure 4.19 (a). Two request tokens for both services are also added in place $Req$. In order to simulate the computation of the computation unit in A, CP-net of invocation connector shown in Figure 4.19 is extended with transitions ($T_{\text{add}}$ and $T_{\text{double}}$).

After simulation, result tokens (for the request tokens from place $Req$) are added into place $Res$ by the CP-net. Service ID 1 represents the add service and service ID 2 represents the double service of component A. The CP-net after simulation is shown in Figure B.2.

Figure B.1: An example in CPN tools

symbol; tokens with different values in a marking are separated by ‘++’ symbol. The current marking of all the places in a net together represents the marking of the net, as shown in Figure B.1(b). Transitions are enabled if there are tokens in the attached input places. When a transition is fired, tokens are moved from the input places to the output places. Transitions can have an optional guard expression.

In a net, arcs can represent flow of tokens and dependency between places and transitions. It is not permitted to have arcs between two places or between two transitions. Tokens can be modified by expressions on the arcs; for example, expression on arc between $T$ and $B$ in Figure B.1 increments the token value.
Figure B.2: Simulation of control/data flows in an invocation connector
B.3 Iloop

In this section, to simulate the control/data flows for an adapted component shown in Figure 4.21(b), tokens are added in place \( c_I \) to represent the provided services by the adapted component. A request token for ‘add’ service is added in place \( Req \). In order to simulate the computation of the connected component, CP-net of iloop connector shown in Figure 4.22 is extended with transition \( aFunc \) which returns empty list for output results.

During simulation response token (from the sub-service request) is added into place \( CP_i \) by transition \( aFunc \) and transition \( T3 \) sends the request to the connected component again. The CP-net during simulation is shown in Figure B.3.
Figure B.3: Simulation of control/data flows in an infinite loop connector
B.4 Sequencer

In this section, to simulate the control/data flows in a sequencer connector of a composite shown in Figure 4.24(a), tokens are added in place $cI$ to represent the provided services by the composite. A request token for addMul service is added in place $Req$. In order to simulate the computation of the composed components, CP-net of sequencer connector shown in Figure 4.25 is extended with transitions ($T_{add}$ and $T_{mul}$).

After simulation response tokens (for the sub-services requests) are added into place $Res$ by transition $T5$. The CP-net after simulation is shown in Figure B.4.
Figure B.4: Simulation of control/data flows in a sequencer connector
B.5 Pipe

In this section, to simulate the control/data flows in a pipe connector for a composite shown in Figure 4.30(a), tokens are added in place $cI$ to represent the provided services by the composite. A request token for ‘addMul’ service is added in place $Req$. In order to simulate the computation of the composed components, CP-net of pipe connector shown in Figure 4.31 is extended with transitions ($T_{\text{add}}$ and $T_{\text{mul}}$).

After simulation response tokens (for the sub-services requests) are added into place $Res$ by transition $T9$. The CP-net after simulation is shown in Figure B.5.
Figure B.5: Simulation of control/data flows in a pipe connector
B.6 Guard

In this section, to simulate the control/data flows for a guard connector in an adapted component shown in Figure 4.38(b), tokens are added in place $cI$ to represent the provided services by the adapted component. A request token for ‘sal’ service is added in place $Req$. In order to simulate the computation of the connected components, CP-net of guard connector shown in Figure 4.40 is extended with transition $T8$.

After simulation response token (for the sub-service request) is added into place $Res$ by transition $T8$. The CP-net after simulation is shown in Figure B.6.
APPENDIX B. COLOURED PETRI NETS

Figure B.6: Simulation of control/data flows in a guard connector
B.7 Selector

In this section, to simulate the control/data flows of a selector connector for a composite shown in Figure 4.48(b), tokens are added in place $cI$ to represent the provided services by the composite. A request token for ‘sal’ service is added in place $Req$. In order to simulate the computation of the composed components, CP-net of selector connector shown in Figure 4.50 is extended with transitions ($T_{Func1}$ and $T_{Func2}$).

After simulation response token (for the sub-service request) is added into place $Res$ by transition $T10$ or by transition $T10'$. The CP-net after simulation is shown in Figure B.7.
Figure B.7: Simulation of control/data flows in a selector connector
In this section, to simulate the control/data flows for a finite loop connector in an adapted component shown in Figure 4.58(b), tokens are added in place $cI$ to represent the provided services by the adapted component. A request token for ‘func’ service is added in place $Req$. In order to simulate the computation of the connected components, CP-net of floop connector shown in Figure 4.60 is extended with transition $T_{func}$ to return false from function $func$.

After simulation, response token (for the sub-service request) is added into place $Res$ by transition $T_{11}$. The CP-net after simulation is shown in Figure B.8.
Figure B.8: Simulation of control/data flows in a finite loop connector
B.9 CoR

In this section, to simulate the control/data flows for a CoR connector in a composite component shown in Figure 6.8, tokens are added in place \( cI \) to represent the provided services by the composite component. A request token for ‘doTask’ service is added in place \( Req \). In order to simulate the computation of the connected component, CP-net of CoR1 connector shown in Figure 6.8 is extended with transitions \( T_{doX} \) and \( T_{doY} \).

After simulation response token (for the sub-service request) is added into place \( Res \) by transitions \( T7 \) or \( T7' \). The CP-net after simulation is shown in Figure B.9.
Figure B.9: Simulation of control/data flows in a CoR connector
B.10  Floop with Parameter Mapping

In this section, to simulate the control/data flows for a finite loop connector (with a constraint for parameter mapping expressions) in an adapted component shown in Figure 6.20(b), tokens are added in place $cI$ to represent the provided services by the adapted component. A request token for ‘func’ service is added in place $Req$. In order to simulate the computation of the connected components, CP-net of floop connector shown in Figure 6.21 is extended with transition $T_{func}$ to return false from function $func$.

After simulation response token (for the sub-service request) is added into place $Res$ by transition $T11$. The CP-net after simulation is shown in Figure B.10.
Figure B.10: Simulation of control/data flows in a finite loop connector with a parameter mapping constraint
Appendix C

Miscellaneous Processes and Algorithms

C.1 Service Signature Extension for Guard

The process of changing service signatures with extra parameters in the service’s constraint of the guard connector is shown in Figure C.1.

\[
\begin{align*}
(1) & \quad \text{pSigs} \xrightarrow{\text{F2}} \text{pSigs}' \\
(2) & \quad \text{pSigs}=\text{pSigs}'.
\end{align*}
\]

Where:

- \(\text{pSigs}' = \{ \text{pSigs} - \{s\} \} \cup \{ \langle s.ID, s.Name, s.IList, s.OList, a.Ind, a.Type, s.Refs \rangle \}
- \text{if s.sID} = c.sID \quad a \in c.conds, \quad a.Ind < \text{|s.sSig.iList|}
- \text{if s.sID} = c.sID \quad a \in c.conds, \quad a.Ind = \text{|s.sSig.iList|}

Function \(F2\) of the connector accepts two inputs of set of provided services of the connector and the set \(Z'\) of parsed service constraints. In the first step, a service’s signature is unchanged by \(F2\) if the service’s constraint is without extra parameters. \(F2\) changes a provided service by appending the extra parameter types (taken from the parsed constraints) to the list of input arguments of a service signature. In a parsed constraint, tuples in \(\text{conds}\) are ordered by the element \(p.Ind\). Hence, an extra parameter is concatenated (shown by the caret symbol in Figure C.1) with the existing list of
input parameters of a service. In the second step, the set of provided services $pSigs$ is updated with the changes.

### C.2 Interface Generation of a Selector Composite

The process of interface generation is shown in Figure C.2.

1. $pSigs \xrightarrow{F1} pSigs'$
2. $pSigs = pSigs'$

Where:

- $1 \leq k \leq 2$, $1 \leq x \leq n$,
- $pSigs \cup \{<y,cSig_x,sSig_{[k]}>\}$, if \( \forall s \in pSigs, s.sSig \neq cSig_x.sSig, \) \( y \in \{\text{positive integers}\} \),
- $pSigs \setminus \{t\} \cup \{<t.sID,t.sSig,t.cList^[k]>\}$, if \( t \in pSigs, t.sSig = cSig_x.sSig, \)

$sSig1 = sSig2$:

- true, if $sSig1.\text{antt} = sSig2.\text{antt}$
- $sSig1.oList = sSig2.oList$ if $\text{true}$
- $sSig1.iList = sSig2.iList$ if $\text{false}$, otherwise.

**Note:** Two lists of parameters (oList/iList) are equal if the order of data types along with their annotations are the same in both lists. Service/parameter annotations (represented by 'antt') are string values.

Figure C.2: The process of interface generation

Initially, set $pSigs$ is empty. In the first step, function $F1$ creates a new set $pSigs'$ from the existing set $pSigs$ in two ways. In the second step, function $F1$ assigns $pSigs'$ to $pSigs$. These two steps execute for each service pair $\langle k, cSig_x \rangle$ ($k$ represents the index of a composed component, $cSig_x$ represents the service signature with index $x$ (a positive integer) of the service in a component) of composed components.

A new service signature (with sID $y$) is added in $pSigs'$ (along with all services of $pSigs$) if a component’s service signature $sSig_x$ is not in $pSigs$. If $F1$ finds a service with signature $sSig_x$ in $pSigs$, the matched service from $pSigs$ is extended (caret symbol is used for this purpose in Figure C.2) to refer to the composed component at index $k$. All provided services of $pSigs$ along with an extended (modified) service are added into $pSigs'$. For two equivalent service signatures, annotations of the two services and the corresponding parameter types along with their annotations (e.g. pre-/post-conditions) from the lists of input/output parameters must be the same.

The provided services set $pSigs$ of a SEL composite are either unmatched services
C.3. SERVICE SIGNATURE EXTENSION FOR SELECTOR

with one value in $cList$ or matched services with two values in $cList$ (having component indices in the connected sequence from left (lower index) to right (higher index)). Matched services of the composed components are referred to as sub-services of the corresponding service in the composite. These services must have an FCL constraint. A service constraint is parsed by the selector as shown in Figure 4.48.

C.3 Service Signature Extension for Selector

For a selector composite, the process of changing service signatures with extra parameters in the service’s constraint is shown in Figure C.3.

Function $F2$ of the connector accepts two inputs of set of provided services of the connector and the set $Z’$ of parsed service constraints. In the first step, a service’s signature is unchanged by $F2$ if the service’s constraint is without extra parameters. $F2$ changes a provided service by appending the extra parameter types (taken from the parsed constraints) to the list of input arguments of a service signature. In a parsed constraint, tuples in $conds$ are ordered by the element $pInd$ as shown in 4.48(b). Hence, an extra parameter is concatenated (shown by the caret symbol in Figure C.3) with the existing list of input parameters of a service. In the second step, the set of provided services $pSigs$ is updated with the changes.

C.4 Interface Generation of a CoR Composite

The process of interface generation is shown in Figure C.4.
(a) pSigs
\[ \text{Where:} \]
cSigs = \{cSig_1, cSig_2, ..., cSig_k\},
cSig = <<sID, sSig>, sID = i>,
pSigs = \{pSig_1, pSig_2, ..., pSig_n\},
pSig = <<sID, sSig, sList>, sID = j>,
sList = [sID_1, sID_2], |Z| = |pSigs|,
sSig = <<sName, oList, iList>, i, j \in \{\text{positive integers}\}.

(b) ∀ a \in cSigs_1, ∀ b \in cSigs_2, pSigs = pSigs \cup \{<j, a.sSig, [a.sID, b.sID]>\}, if a = b ∧ |a.iList| = 0 ∧ |a.oList| ≠ 0,
\[ \text{Where:} j \text{ represents a positive integer,} \]
\[ \text{two lists of parameters (oList/iList) are equal if the order of data types along with their annotations} \]
\[ \text{are the same in both lists, service/parameter annotations (represented by 'antt') are string values.} \]

\[ \text{Figure C.4: Interface generation} \]

For a matched service (with the empty list of input parameters and with the non-empty list of output parameters) from the composed components, a compound service (with the reference of matched services from the composed components) is added into the provided interface of the composite.
Appendix D

Cash Desk System Diagrams

Figure D.1: \( S_0, S_1, S_2, S_3, \) and \( S_4 \)
connection points are placed to avoid cluttering

Figure D.2: $S_5$, $S_6$, $S_7$, and $S_8$
Figure D.3: $S_9$ and $S_{10}$

Figure D.4: $S_{11}$ and $S_{12}$
Figure D.5: $S_{13}$, $S_{14}$, and $S_{15}$
Figure D.6: $S_{16}$ and $S_{17}$
Figure D.7: $S_{18}$ and $S_{19}$
Figure D.8: $S_{20}$, and $S_{21}$
Figure D.9: $S_{22}$, and $S_{23}$
Figure D.10: $S_{24}$