CONSTRUCTING COMPONENT-BASED SYSTEMS DIRECTLY FROM REQUIREMENTS USING INCREMENTAL COMPOSITION

A dissertation submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2013

By Azlin Nordin
School of Computer Science
2.2.4 Category 4: Deal with a single requirement and build the system architecture in each incremental step ........ 72
2.2.5 The Proposed Approach .............................. 73
2.3 Summary .................................................. 74

3 The X-MAN Component Model .......................... 75
3.1 Component Models ...................................... 75
3.2 What is a Component Model? ............................. 77
  3.2.1 Semantics ............................................ 77
  3.2.2 Syntax ............................................... 77
  3.2.3 Composition ........................................ 78
3.3 Component Life Cycle .................................... 78
  3.3.1 Design Phase ........................................ 78
  3.3.2 Deployment Phase ................................... 80
  3.3.3 Run-Time Phase ..................................... 80
3.4 The X-MAN Component Model ........................... 81
3.5 Key Elements of X-MAN ................................. 84
  3.5.1 Computations ........................................ 84
  3.5.2 Control .............................................. 84
  3.5.3 Composition ........................................ 90
3.6 Intrinsic Properties of the Component Model .......... 91
  3.6.1 Component Encapsulation ............................ 92
  3.6.2 Compositionality .................................... 92
  3.6.3 Component Reusability ............................... 93
3.7 A Simple Bank System Example ........................ 94
3.8 Summary .................................................. 95

4 Incremental Composition ............................... 96
4.1 Introduction ............................................. 96
4.2 A Component Model with Incremental Composition .... 99
4.3 Incremental Composition in Existing Component Models .... 102
  4.3.1 Architecture Description Languages ................. 102
  4.3.2 Aspect-Oriented Composition ........................ 103
  4.3.3 Invasive Software Composition ....................... 104
  4.3.4 Behaviour Engineering ............................... 105
4.4 Design Decisions for Incremental Composition ......... 107
4.4.1 Ordering-based Composition ........................................ 109
4.4.2 Selection-based Composition ........................................ 113
4.4.3 Condition-based Adaptation ......................................... 116
4.4.4 Repetition-based Adaptation ......................................... 119

4.5 Issues and Discussion ................................................... 120
4.5.1 Combining redundant behaviours ................................... 121
4.5.2 Amendment of constraints involved during IC .................. 121
4.5.3 Consideration of piped data .......................................... 121

4.6 Summary ................................................................. 122

5 Extracting Elements of CB Systems ................................... 123
5.1 Introduction ............................................................. 124
5.2 Related Work on Information Extraction ............................ 124
5.3 Identifying Component-based Elements From NLR ............... 125
5.3.1 Elements Extraction from Natural Language Requirements 126
5.3.2 Identifying Component-based Constructs of the X-MAN Component Model ..................................................... 130
5.4 Example ................................................................. 148
5.4.1 A simplified Automated Teller Machine (ATM) system .... 149
5.5 Implementation of the Extractor Tool ............................... 155
5.5.1 Application of a Part-Of-Speech Tagger ......................... 156
5.5.2 Element Extraction Using the Extractor Tool ................. 159
5.6 Issues and Discussion ................................................... 160
5.6.1 Requirements Problems .............................................. 161
5.6.2 Dealing with Implicit Requirements ............................... 161
5.6.3 Issues on Computation Identification ............................ 161
5.6.4 Handling More Than One Control Extraction .................. 162
5.6.5 Limitations of the Elements Extraction .......................... 162
5.7 Summary ................................................................. 163

6 Mapping from Reqs. to Arch. Elements .............................. 164
6.1 The Mapping Process ................................................... 164
6.1.1 Overview ............................................................. 165
6.1.2 The Mapping Definition ........................................... 167
6.2 Mapping Function RQ to EE .......................................... 171
8.2.4 Creating partial architectures ............................................ 231
8.2.5 Composing partial architectures with the existing system architecture ............................................ 232
8.2.6 Finalising the system architecture ............................................ 236
8.3 A Complete Example: The ATM System ............................................ 238
  8.3.1 Increment-1 - Requirement 1 ............................................ 238
  8.3.2 Increment-2 - Requirement 2 ............................................ 241
  8.3.3 Increment-3 - Requirement 3 ............................................ 245
  8.3.4 Increment-4 - Requirement 4 ............................................ 247
  8.3.5 Increment-5 - Requirement 5 ............................................ 249
  8.3.6 Increment-6 - Requirement 6 ............................................ 251
  8.3.7 Increment-7 - Requirement 7 ............................................ 253
  8.3.8 Step-5: Finalising the system architecture ............................................ 255
8.4 Validation of the Incremental Approach ............................................ 256
  8.4.1 The Trading System ............................................ 257
  8.4.2 Other Case Studies ............................................ 266
8.5 Issues and Discussion ............................................ 268
  8.5.1 Requirements Problem ............................................ 269
  8.5.2 Element Extraction Exceptions ............................................ 269
  8.5.3 Component Selection ............................................ 269
  8.5.4 Validation of the Derived Architecture ............................................ 269
8.6 Summary ............................................ 270

9 Evaluation and Discussion ............................................ 271
  9.1 Introduction ............................................ 271
  9.2 Preliminary Empirical Validation ............................................ 272
    9.2.1 Objectives ............................................ 272
    9.2.2 Research Questions ............................................ 273
    9.2.3 Instrumentation and Materials ............................................ 274
    9.2.4 Pilot Study ............................................ 281
    9.2.5 The Main Experiment ............................................ 293
    9.2.6 Threats to Validity ............................................ 305
  9.3 Summary ............................................ 306
  9.4 Analysis of the Case Studies ............................................ 306
  9.5 Reflections on the Approach ............................................ 307
    9.5.1 Properties of the X-MAN Component Model ............................................ 307
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5.2 Requirement Issues</td>
<td>311</td>
</tr>
<tr>
<td>9.5.3 The Effects of the Selected Properties of the Incremental</td>
<td>313</td>
</tr>
<tr>
<td>Approach</td>
<td></td>
</tr>
<tr>
<td>9.5.4 The Effects of the Tools Support</td>
<td>314</td>
</tr>
<tr>
<td>9.6 Comparison with Related Work</td>
<td>316</td>
</tr>
<tr>
<td>9.7 Issues and Discussion</td>
<td>317</td>
</tr>
<tr>
<td>9.7.1 Support for Software Development Project</td>
<td>317</td>
</tr>
<tr>
<td>9.7.2 Requirements authoring styles</td>
<td>320</td>
</tr>
<tr>
<td>9.7.3 Potential Effects of Requirements Ordering, Dependencies</td>
<td>321</td>
</tr>
<tr>
<td>and Prioritisation</td>
<td></td>
</tr>
<tr>
<td>9.7.4 Achieving requirements satisfaction</td>
<td>324</td>
</tr>
<tr>
<td>9.7.5 The Resulting Architecture</td>
<td>325</td>
</tr>
<tr>
<td>9.8 Summary</td>
<td>325</td>
</tr>
<tr>
<td>10 Conclusions and Future Work</td>
<td>326</td>
</tr>
<tr>
<td>10.1 Research Contribution</td>
<td>326</td>
</tr>
<tr>
<td>10.2 Limitations and Discussion</td>
<td>328</td>
</tr>
<tr>
<td>10.3 Future Work</td>
<td>329</td>
</tr>
<tr>
<td>10.3.1 The Use of NLP</td>
<td>330</td>
</tr>
<tr>
<td>10.3.2 Execution of a Large-scale Set of Requirements</td>
<td>330</td>
</tr>
<tr>
<td>10.3.3 An Experimentation of the Cost and Benefit of the Approach</td>
<td>330</td>
</tr>
<tr>
<td>10.3.4 Architecture Refactoring</td>
<td>331</td>
</tr>
<tr>
<td>10.3.5 Automation of the Architecture Refactoring</td>
<td>331</td>
</tr>
<tr>
<td>10.3.6 An Integrated Tool Support</td>
<td>332</td>
</tr>
<tr>
<td>10.4 Summary</td>
<td>332</td>
</tr>
<tr>
<td>Bibliography</td>
<td>333</td>
</tr>
<tr>
<td>A Textual Analysis</td>
<td>362</td>
</tr>
<tr>
<td>A.1 Introduction</td>
<td>362</td>
</tr>
<tr>
<td>A.2 Part-of-Speech Tagger</td>
<td>363</td>
</tr>
<tr>
<td>A.2.1 What is a POS Tagger?</td>
<td>363</td>
</tr>
<tr>
<td>B Questionnaire</td>
<td>366</td>
</tr>
<tr>
<td>C Case Studies</td>
<td>378</td>
</tr>
<tr>
<td>C.1 The Trading System (COCOME)</td>
<td>379</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of the Extracted Keywords from R1</td>
<td>151</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of the Extracted Keywords from R2</td>
<td>151</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary of the Extracted Keywords from R3</td>
<td>152</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of the Extracted Keywords from R4</td>
<td>153</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of Extracted Keywords from R5</td>
<td>154</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary of the Extracted Keywords from R6</td>
<td>154</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary of Extracted Keywords from R7</td>
<td>155</td>
</tr>
<tr>
<td>8.1</td>
<td>Summary of Steps 1 and 2 for R1</td>
<td>240</td>
</tr>
<tr>
<td>8.2</td>
<td>Summary of Steps 1 and 2 for R2</td>
<td>243</td>
</tr>
<tr>
<td>8.3</td>
<td>Summary of Steps 1 and 2 for R3</td>
<td>246</td>
</tr>
<tr>
<td>8.4</td>
<td>Summary of Steps 1 and 2 for R4</td>
<td>248</td>
</tr>
<tr>
<td>8.5</td>
<td>Summary of Steps 1 and 2 for R5</td>
<td>250</td>
</tr>
<tr>
<td>8.6</td>
<td>Summary of Steps 1 and 2 for R6</td>
<td>252</td>
</tr>
<tr>
<td>8.7</td>
<td>Summary of Steps 1 and 2 for R7</td>
<td>254</td>
</tr>
<tr>
<td>8.8</td>
<td>Summary of the Executed Case Studies and Examples</td>
<td>266</td>
</tr>
<tr>
<td>9.1</td>
<td>Summary of Extraction Scoring Criteria</td>
<td>277</td>
</tr>
<tr>
<td>9.2</td>
<td>Summary of Mapping Scoring Criteria for Correctness</td>
<td>278</td>
</tr>
<tr>
<td>9.3</td>
<td>Summary of Mapping Scoring Criteria for Syntax</td>
<td>279</td>
</tr>
<tr>
<td>9.4</td>
<td>Extraction Scoring for Pilot Study</td>
<td>284</td>
</tr>
<tr>
<td>9.5</td>
<td>Summary of Correctness Scoring Value</td>
<td>285</td>
</tr>
<tr>
<td>9.6</td>
<td>Summary of Scoring Value for Syntax</td>
<td>287</td>
</tr>
<tr>
<td>9.7</td>
<td>Adoption of the Extraction and Mapping Heuristics</td>
<td>288</td>
</tr>
<tr>
<td>9.8</td>
<td>Extraction Scoring</td>
<td>296</td>
</tr>
<tr>
<td>9.9</td>
<td>Summary of Scoring Value for Correctness</td>
<td>297</td>
</tr>
<tr>
<td>9.10</td>
<td>Scoring Value for Syntax</td>
<td>299</td>
</tr>
<tr>
<td>9.11</td>
<td>Adoption of the Extraction and Mapping Heuristics</td>
<td>301</td>
</tr>
<tr>
<td>9.12</td>
<td>Perceived Difficulty Level for Training</td>
<td>303</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Conceptual model. .............................................. 41
1.2 Relationships Between Chapters. ............................ 46

2.1 Current scenario .................................................. 48
2.2 Summary of Existing Mapping Approaches .................... 51
2.3 Category 1 .......................................................... 52
2.4 ADEF [LE03] ..................................................... 53
2.5 Relationships between scenarios, requirements and architectures [ZMP03] .................................................. 54
2.6 Simplified EDS Process Model [BBG00] ....................... 55
2.7 Scenario Meta Model [PBG01] .................................. 57
2.8 Category 2 .......................................................... 59
2.9 Feature Model ..................................................... 60
2.10 Feature Mapping [LM03] ......................................... 61
2.11 The Twin Peaks Model [Bas01] ............................... 63
2.12 CBSP Conceptual Framework [GEM04] ....................... 64
2.13 CBSP Process [GEM04] .......................................... 65
2.14 GORE Process ................................................... 68
2.15 SAORE [Mei00] .................................................. 69
2.16 Category 3 .......................................................... 70
2.17 Behaviour Trees .................................................. 71
2.18 Category 4 .......................................................... 72

3.1 Component Life Cycle ............................................. 78
3.2 X-MAN component model. ...................................... 82
3.3 Composition connectors ........................................ 82
3.4 Hierarchical composition by a SEQ ............................ 83
3.5 Component and Composition .................................... 85
3.6 Invocation Connector (IC) ....................................... 86
3.7 Generic Composition Connectors .................................. 87
3.8 Adaptation Operators ............................................. 89
3.9 Simple Bank System Example ..................................... 94

4.1 Composition in C2 ..................................................... 97
4.2 Incremental composition ............................................ 98
4.3 The Basic Component Model Elements ............................ 99
4.4 Incremental Composition in X-MAN ................................. 101
4.5 Generic ADLs .......................................................... 103
4.6 Aspect-Oriented Composition ...................................... 103
4.7 Invasive Software Composition ................................... 104
4.8 The Result of a Sequencing Composition ......................... 105
4.9 The Result of a Branching Composition ............................ 106
4.10 Composition points ................................................. 108
4.11 Adding a component to an existing composition ............... 110
4.12 ATM example-1 ....................................................... 111
4.13 Word Count-2 ......................................................... 112
4.14 ATM Example-3 ....................................................... 112
4.15 ATM Example-4 ....................................................... 113
4.16 ATM Example-5 ....................................................... 114
4.17 Composing to a new component to a composition point .... 114
4.18 ALCS ................................................................. 115
4.19 Incremental Composition for Selection-based Composition .... 116
4.20 Add a Guard to a Composition Point .............................. 117
4.21 Add a Guard to a Composition Point .............................. 118
4.22 Adding a Guard ....................................................... 118
4.23 Adding a loop adapter ............................................... 119
4.24 Looping example ..................................................... 120
4.25 Repetition-based composition .................................... 120

5.1 Related work on Information Extraction ......................... 124
5.2 Elements that can be extracted from verbs ...................... 127
5.3 Elements that can be extracted from nouns ...................... 128
5.4 Elements that can be extracted from phrases .................... 129
5.5 Extracting computations ............................................ 131
5.6 Extracting control .................................................... 140
7.14 Pipe with Guards Pattern .............................................. 211
7.15 Exceptional cases — VSS ............................................. 212
7.16 Before and after refactoring .......................................... 213
7.17 Refactor guards — WordCount ....................................... 214

8.1 Research Context — A Conceptual model .............................. 220
8.2 The Incremental Approach .............................................. 220
8.3 The Flow of the Extractor Tool ........................................ 228
8.4 The Extractor Tool ...................................................... 229
8.5 The Exogenous Composition Framework Tool .......................... 235
8.6 Finalising Composition Points ......................................... 236
8.7 Adding a Loop Adapter .................................................. 237
8.8 Extraction of R1 using the Extractor Tool ............................. 239
8.9 Partial Architecture for R1 .............................................. 241
8.10 Extraction of R2 using the Extractor Tool ............................. 242
8.11 Increment-2 ............................................................ 244
8.12 Increment-3 ............................................................ 247
8.13 Increment-4 ............................................................ 249
8.14 Increment-5 ............................................................ 251
8.15 Increment-6 ............................................................ 253
8.16 Increment-7 ............................................................ 255
8.17 Final System Architecture .............................................. 256
8.18 COCOME architecture ................................................... 258
8.19 X-MAN composition for COCOME .................................... 259
8.20 Test cases execution — TC-UC1A ...................................... 261
8.21 Test cases execution — TC-UC1B ...................................... 261
8.22 Test case execution - TC-UC1C ....................................... 262
8.23 Test cases execution — TC-UC [1D, 1E] .............................. 263
8.24 Test cases execution — TC-UC [3A, 4A] .............................. 264
8.25 Test cases execution — TC-UC3A ..................................... 264
8.26 Test cases execution — TC-UC [6A, 6B] .............................. 265
8.27 Test cases execution — TC-UC [7A, 7B] .............................. 266
8.28 Final System Architecture for the Steam Boiler System ......... 268

9.1 An example of a Questionaire Item for Extraction Heuristics ..... 280
9.2 An example of a Questionaire Item for Mapping Heuristics ..... 280
<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.21 Part of the incremented architecture for UC3-R4</td>
<td>410</td>
</tr>
<tr>
<td>C.22 UC4-R3</td>
<td>412</td>
</tr>
<tr>
<td>C.23 UC4-R4</td>
<td>413</td>
</tr>
<tr>
<td>C.24 UC5-R1</td>
<td>415</td>
</tr>
<tr>
<td>C.25 Part of the incremented architecture for UC5-R2</td>
<td>416</td>
</tr>
<tr>
<td>C.26 UC6-R1</td>
<td>417</td>
</tr>
<tr>
<td>C.27 Part of the incremented architecture for UC6-R2</td>
<td>418</td>
</tr>
<tr>
<td>C.28 UC7-R1</td>
<td>419</td>
</tr>
<tr>
<td>C.29 Part of the incremented architecture for UC7-R3</td>
<td>420</td>
</tr>
<tr>
<td>C.30 UC8-R3</td>
<td>422</td>
</tr>
<tr>
<td>C.31 UC8-R4</td>
<td>423</td>
</tr>
<tr>
<td>C.32 Incremented architecture for UC8-R5</td>
<td>423</td>
</tr>
<tr>
<td>C.33 Incremented architecture for UC8-R7</td>
<td>425</td>
</tr>
<tr>
<td>C.34 Incremented architecture for UC8-R8</td>
<td>426</td>
</tr>
<tr>
<td>C.35 Incremented architecture for UC8-E1</td>
<td>427</td>
</tr>
<tr>
<td>C.36 Incremented architecture for UC8-Ext3</td>
<td>429</td>
</tr>
<tr>
<td>C.37 COCOME Architecture Before Refactoring</td>
<td>430</td>
</tr>
<tr>
<td>C.38 Architecture Refactoring</td>
<td>430</td>
</tr>
<tr>
<td>C.39 AR-1</td>
<td>431</td>
</tr>
<tr>
<td>C.40 AR-2</td>
<td>432</td>
</tr>
<tr>
<td>C.41 AR-3</td>
<td>432</td>
</tr>
<tr>
<td>C.42 AR-4</td>
<td>432</td>
</tr>
<tr>
<td>C.43 AR-5</td>
<td>433</td>
</tr>
<tr>
<td>C.44 AR-6</td>
<td>433</td>
</tr>
<tr>
<td>C.45 COCOME Architecture After Refactoring</td>
<td>434</td>
</tr>
<tr>
<td>C.46 Finalised Architecture</td>
<td>434</td>
</tr>
<tr>
<td>C.47 Result of mapping for R1</td>
<td>437</td>
</tr>
<tr>
<td>C.48 R4</td>
<td>438</td>
</tr>
<tr>
<td>C.49 R5</td>
<td>439</td>
</tr>
<tr>
<td>C.50 R6</td>
<td>440</td>
</tr>
<tr>
<td>C.51 R8</td>
<td>441</td>
</tr>
<tr>
<td>C.52 Part of the Incremented Architecture for R9</td>
<td>442</td>
</tr>
<tr>
<td>C.53 Part of the Incremented Architecture for R10</td>
<td>442</td>
</tr>
<tr>
<td>C.54 Part of the Incremented Architecture for R11</td>
<td>443</td>
</tr>
<tr>
<td>C.55 R12</td>
<td>444</td>
</tr>
<tr>
<td>C.56</td>
<td>R13</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
</tr>
<tr>
<td>C.57</td>
<td>R14</td>
</tr>
<tr>
<td>C.58</td>
<td>R15</td>
</tr>
<tr>
<td>C.59</td>
<td>R16</td>
</tr>
<tr>
<td>C.60</td>
<td>R17</td>
</tr>
<tr>
<td>C.61</td>
<td>R18</td>
</tr>
<tr>
<td>C.62</td>
<td>R19</td>
</tr>
<tr>
<td>C.63</td>
<td>Incremented Architecture for R19</td>
</tr>
<tr>
<td>C.64</td>
<td>VSS Architecture Before Refactoring</td>
</tr>
<tr>
<td>C.65</td>
<td>Label for VSS Architecture Refactoring</td>
</tr>
<tr>
<td>C.66</td>
<td>Refactored VSS Architecture</td>
</tr>
<tr>
<td>C.67</td>
<td>Finalised VSS Architecture</td>
</tr>
<tr>
<td>C.68</td>
<td>Result of Mapping for R2</td>
</tr>
<tr>
<td>C.69</td>
<td>Partial and Incremental Architecture for R3</td>
</tr>
<tr>
<td>C.70</td>
<td>Partial, Incremented and Updated Architecture for R4</td>
</tr>
<tr>
<td>C.71</td>
<td>Partial, Incremented and Updated Architecture for R5</td>
</tr>
<tr>
<td>C.72</td>
<td>Refactoring for WC.</td>
</tr>
<tr>
<td>C.73</td>
<td>Finalisation Step.</td>
</tr>
</tbody>
</table>
List of Heuristics and Design Decisions

5.1 Identifying Computations from Text .................................. 132
5.2 Identifying Computations from User Interaction .................. 133
5.3 Identifying Conceptual Components ................................. 139
5.4 Identifying Control .................................................... 145
5.5 Identifying Data ......................................................... 147
6.1 Mapping Computation Keywords to Component’s Computations .. 179
6.2 Mapping Control Keywords to Composition Connectors .......... 182
6.3 Mapping Control Keywords to Adapters ............................ 189
6.4 Mapping Data Keywords to Data Elements .......................... 189
7.1 Combining Homogeneous Composition Connectors ............... 202
7.2 Adding a Lower-level Hierarchy ..................................... 208
7.3 Adding an Upper-level Hierarchy .................................... 209
7.4 PIPE with Guards Pattern .......................................... 214
Abstract

In software engineering, system construction typically starts from a requirements specification that has been engineered from raw requirements in a natural language. The specification is used to derive intermediate requirements models such as structured or object-oriented models. Throughout the stages of system construction, these artefacts will be used as reference models. In general, in order to derive a design specification out of the requirements, the entire set of requirements specifications has to be analysed. Such models at best only approximate the raw requirements since these design models are derived as a result of the abstraction process according to the chosen software development methodology, and subjected to the expertise, intuition, judgment and experiences of the analysts or designers of the system. These abstraction models require the analysts to elicit all useful information from the requirements, and there is a potential risk that some information may be lost in the process of model construction.

As the use of natural language requirements in system construction is inevitable, the central focus of this study was to use requirements stated in natural language in contrast to any other requirements representation (e.g. modelling artefact). In this thesis, an approach that avoids intermediate requirements models, and maps natural language requirements directly into architectural constructs, and thus minimises information loss during the model construction process, has been defined. This approach has been grounded on the adoption of a component model that supports incremental composition. Incremental composition allows a system to be constructed piece by piece. By mapping a raw requirement to elements of the component model, a partial architecture that satisfies that requirement is constructed. Consequently, by iterating this process for all the requirements, one at a time, the incremental composition to build the system piece by piece directly from the requirements can be achieved.
Declaration

No portion of the work referred to in this dissertation 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://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf), 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.
Supplementary Material

This thesis contains accompanying material in the form of electronic resources included in an attached compact disc. The followings are the contents of the electronics resources:

1. Publications:


2. Case Studies: Electronic copies of all the case studies used in this thesis in Portable Document Format (PDF) format.

   (a) The Trading System (COCOME)

   (b) The Video Store System (VSS)

   (c) The Word Count Example
Acknowledgements

This thesis reflects what has been achieved during this four-year PhD period as a result of the learning process and the research skill experiences. I would like to gratefully acknowledge my family, parents, colleagues and friends who have directly or indirectly contributed towards my PhD journey, without which this journey might not have come to the end.

First and foremost, I would like to thank my supervisor, Dr. Kung-Kiu Lau, for all his efforts in supervising me throughout this PhD period. His advice, criticisms and insights into research have enlightened me with the required research skills and experiences. Also thank you for all feedback during the series of group workshops throughout these years. As a life-time investment and a stepping stone for my future career, this is such a valuable personal experience, which cannot easily be achieved without much effort and pain. I would also like to thank the examiners for agreeing to review this thesis.

I am particularly grateful to my understanding husband, Ahmad Fadilah Ahmad and my son, Ahmad Irfan, for supporting me during these difficult yet challenging years. Living thousands of miles away from home in new surroundings, the adaptation process has been easier with the ones you need with you. Thank you for all the difficult sacrifices and the substantial times that have been spent in joining with me during this PhD battle. I am also indebted to both my loving parents, Nordin Ahmad and Zabedah Hj. Mansor for the support and prayers
during all these years.

I also wish to extend my gratitude to my colleagues in the group at (1) the School of Computer Science, University of Manchester: Lily, Ng Keng Yap, Tauseef, Yannis, Faris, Cuong and Petr; (2) the Kulliyyah of Information and Communication Technology (KICT), International Islamic University of Malaysia (IIUM): Lili Marziana, Sofianiza, Norsaremah and Amelia. Special thanks go to Lily and Nurul Akmar, for being supportive and understanding with my ups and downs all these years. In addition, thanks to Tauseef and Ng Keng Yap for the collaboration opportunities that have resulted in publications. I will always cherish the times we spent all these years in the office and thank you for being such nice colleagues and friends.

Finally, I would like to acknowledge the Malaysian Ministry of Higher Education and IIUM for financial assistance and sponsorship during this PhD period.

Thank you, from the bottom of my heart.
Chapter 1

Introduction

In Software Engineering (SE), one of the ultimate aims of software development is to build systems that satisfy user requirements. Nevertheless, achieving such an aim is obviously a challenging task. The software development process starts with the elicitation of requirements, and follows with the analysis and design stages. The design is then implemented and tested. In the literature, there are various software development methodologies and Requirements Engineering (RE) methods, as well as techniques that address various ways of improving the software development process. The traditional example of the software development methodology is the Waterfall Model [Roy87], where the process is sequential, and a system design consists of modules that interact with one another. A modern example is the Unified Development Process [JBR99], where the process is iterative, and a system design consists of classes that interact with one another (represented by UML class and interaction diagrams).

Component-based Software Engineering (CBSE), on the other hand, is another specialised area of SE that deals with achieving software component reusability. To be more specific, CBSE as defined in [Som07, p. 440], is “the process of
defining, implementing and integrating or composing loosely coupled independent components into systems”. A software component is defined as “a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard” [HC01]. For component-based systems, it may be questioned whether CBSE should adopt different software development methodologies going from requirements to the analysis to the final system, in contrast to conventional software development. The conventional software development process is viewed as restricting the potential of the CBSE [Som07, p. 441]. The main reason is that, a component-based system should be constructed using a component model [LW07] that defines what are components, its syntaxes and semantics, and how the components should be composed.

In addition to the general challenges in conventional software development, CBSE has its own distinct challenges, in contrast to conventional software development approaches. Among these challenges are component trustworthiness, certification, emergent property prediction [Som07, p. 441] and reuse of available components or acquisition of the Component-Off-The-Shelf (COTS) components from the market [KH07]. In considering these challenges in the CBSE domain, it is not surprising that CBSE is viewed as relatively less desirable in software development as compared to the other conventional or emergent paradigms (e.g. structured, object-oriented, aspect-oriented). Beyond this, however, the benefits that CBSE offers (i.e. reusability, encapsulation, and compositionality) should extend thoughtful consideration over these limitations. This indicates a need to better understand these situations and look into better solutions, in an attempt to realise and improve the CBSE processes.

In order to build component-based systems, generally, a requirements engineer or an analyst starts with the analysis of the requirement documents. The
fact that most of these requirements documents are written in natural language is irrefutable [BK00]\(^1\) as this artefact is used as the main communication tool between users and the developer’s team. The use of natural language requirements (NLR) will never be totally replaced by formal specification and hence, the need to accommodate them is undeniable [Rut77].

RE deals with requirements in the upfront activities in software development. This area is considered as one of the key activities in software development. It is, in fact, motivated by the view that requirements problems are much more expensive to fix later in the development cycle [Boe84, Gla03]. Requirements problems with regard to satisfying the real needs of the user have been reported in many articles [DvE06, KS03, Ram98, EEM95, RJ01]. For decades, this has been an open issue in SE research. Although it is clear that in conventional SE, namely structured and object-oriented approaches, the aim of the development is to satisfy the user requirements, this intention is, in fact, led by many factors. Glass [Gla03] indicated that the most typical cause of delinquent projects is unstable requirements. Having said this, as most software developments are dependent on the requirements, beyond doubt developing software to satisfy the user requirements is a huge challenge.

This chapter sets the scene with a brief introduction to RE and continues with some insights into the problem description. The following sections present the research objectives, an overview of the incremental approach, scope and limitations, and the research contributions. Finally, the thesis structure is presented.

\(^1\)RE is where the informal meets the formal [Jac95].
1.1 Background

Before going into further details, it is worthwhile providing some fundamental background on RE. The RE process is “a structured set of activities, which are used to derive, validate and maintain a systems requirements document” [KS03]. This RE process is a multi-disciplinary activity, deploying a variety of techniques and tools at different stages of software development and for different kinds of application domains [NE00]. In other words, Zave [Zav97] defines RE as “the branch of SE concerned with the real-world goals for functions of and constraints on software systems”. Recently, contributions from the RE field have become significantly accepted among the SE community, and as a result of this, the importance of RE has been greatly emphasised in the literature (e.g. [EEM95, KS03, Ram98, RJ01, KDR+07, HL01, ABE+06]).

The RE process is “a structured set of activities, which are used to derive, validate and maintain a systems requirements document” [KS03, p. 9]. The core RE activities include elicitation, analysis and modelling, communication, validation and verification, documentation and evolving requirements [NE00, PA06, Som07]. The RE activities primarily define the choice of RE method to be adopted. RE for information systems, for instance, is performed differently from RE for embedded control systems [NE00]. These activities are mostly performed iteratively and incrementally until all the RE tasks can be plausibly applied to further SE processes. As supported by SWEBOK [IEE04], requirements are expected to be iterative to a level of quality and detail which is sufficient to allow design decisions to be made.

In general, the literature, explicitly or implicitly, describes requirements as either user or system requirements. Few definitions and types of requirement are presented. SWEBOK presents the general definition of a requirement as “a
1.1. BACKGROUND

property, which must be exhibited in order to solve some problem in the real world” [IEE04]. Another general definition of a requirement is “something that the product must do or a quality that the product must have” [RR99]. In another study, a requirement is defined as “an expression of desired behavior, which deals with objects or entities, the states they can be in, and the functions that are performed to change states or object characteristics” [PA06].

In the literature, the requirements can generally be classified into functional and non-functional (NFR) or quality requirements [RR99, Sch07, IEE04, Som07]. Although there are various definitions of ‘requirements’ in the literature and software industry (e.g. [RR99, PA06, IEE04, Sch07, Som07, KS03]), the raw requirements, which are of interest in this work, are confined to requirements that state observable behaviours or functional requirements only. A functional requirement is “a statement of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations, and sometimes explicitly state what the system should not do” [Som07].

Once the analysis stage is completed, the following stage is that of design. During the design stage, one of the activities is to produce software architecture (SA). A generic definition of SA is “the program structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” [BCK03]. The authors also agree that the behaviours of each element are part of the architecture. In relating to this point, behaviours of an element can be observed from the perspective of another element. Hence, behaviours should be represented as a part of the architecture. In this research, the term SA can generally be used as an abstraction of element information, and the communication between these elements. In this report, the elements here are regarded as components.
1.2 Problem Description and Motivation

The application of RE processes in software development have been seen to have a significant effect on the success of software development projects. This reflects the importance of the RE processes in any software development, in order to achieve user satisfaction with the system. Recent studies in software development have heightened the need to enhance the quality of software delivered to the client by ensuring that user requirements are satisfied, risks of errors are reduced as the requirements and the domains of the applications become increasingly complex.

The past decade has seen rapid development among RE and CBSE researchers in defining alternative ways of developing software, thus gaining benefits from both the areas. To date, most studies in RE have tended to focus on the RE processes themselves rather than their relationships to other related software development processes. However, little attention has been paid to bridging these gaps, in relation to architecture as indicated in the literature [DC06, GEM06, FM08, GEM04, GEM01, Bas05, vL03].

Several challenges [GEM04] with regard to these gaps are identified in this thesis including:

(a) Representation of requirements that is generally in natural language. This implies that the requirements are documented in an informal manner, whilst SA documents are specified in a semi-formal or more formal way.

(b) The nature and evolution of requirements, reflecting the fact that architecture development should withstand the requirements risks being incomplete and in a state of evolution. Furthermore, some of the requirements can only be fully understood after the system architecture has been developed.
1.2. **PROBLEM DESCRIPTION AND MOTIVATION**

(c) Traceability issues when maintaining both the requirements and architecture are non-trivial.

(d) The size of real world, large-scale software requirements makes it difficult to identify and update the required information into the related architectural design.

The above-listed challenges indicate the complexity involved in dealing with the tasks of reducing the gaps between these two stages. In addition, an interesting insight in the literature states that the existence of these gaps is due to the fact that different concepts and terms are used to represent the required models, which are relevant in both the stages [GEM04]. Hence, it remains an intriguing challenge to attempt to address these gaps.

While a considerable amount of RE literature discusses the application of RE in the early phases of the software development life cycle, nonetheless the relevance of RE to other software processes is broadly acknowledged in the SE community. A current trend in RE literature seeks to investigate the relationships between these stages, and few approaches have been proposed to reduce these gaps. However, a study [Bas05] states that the connection between RE and architecture is not very well understood and therefore a restricted direction is applicable. It is also mentioned that although there are gaps between these two stages, a systematic way of dealing with these gaps is needed [Bas05].

In spite of the fact that investigating the gaps between RE and architectures is acknowledged in the literature, to date, SE researchers do not have a consensus on the connection of these two areas [LM03]. In previous research, impact studies of RE on software development have been undertaken independently, thereby separating the concerns of RE processes and their impact on software development. In order to address this issue, an empirical study [DC06] explores the
relationships between RE and other software development processes. The outcome of the research demonstrates the positive impact of RE in the perspectives of productivity, quality and risk management in software development.

It is becoming increasingly difficult to ignore the importance of reducing the gaps from requirements to architectures. Having said this, understanding and investigating these gaps has been one of the emerging research areas in RE. This has also been reflected in a study [FE00] where the prospects for future Requirements Management (RM) tools for the short, medium and long term are presented, highlighting the fact that no attention has been paid to the relationship between requirements and architectures. As a result of the study, the relationship between requirements and architectures becomes one of the desired features of RM tools in the long-term category.

The advantages of reducing the gaps between the phases have been discussed in the literature (e.g. [Bas05, BBGM00, HNS05, GEM04]). The prominent issue in transforming requirements into an architecture and further into software models or artefacts is the issue of traceability. Traceability as defined by IEEE [IEE90] is “the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another”. One of the early works on requirements traceability (RT) refers to RT as “the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases” [GF94].

Moreover, standards, (i.e. MIL-STD-498 [Dep94, ROC95] and IEEE/EIA J-STD-016 [Sor96]), and assessment (i.e. Capability Maturity Model Integration
(CMMI) [Tea06]) have also affirmed that RT activities should be properly documented. For these reasons, traceability of the software artefacts is important in any software development, as it demonstrates whether the product satisfies all the elicited requirements, and thus represents what the stakeholders really want, and eventually becomes a benchmark for the quality of the product. Although its central importance as a key success factor in software development has been highlighted in many items of literature (e.g. [HL01, You06, Fir07, Ram98, RJ01, EEM95, BB01, BH07, HDSH04, KDR^07, SGK^04]), RT remains in the state where its primary aim to capture the relationships between important software artefacts is not fully attained due to many contributing factors. By having transparent transitions from requirements to architectures, it can inherently contribute to handling this traceability issue.

Recent trends have shown an increased interest in exploring and investigating the relationships between requirements and architectures. This is also reflected and cited as one of the challenges in SE literature (i.e. [Bas05, NE00, NE00, GEM04]). One of the most significant current discussions in highlighting this issue is discussed in a survey [GEM06]. This survey describes, categorises and examines existing approaches (i.e. Architecture Description Language (ADL), Goal-based Approach, Problem Frames, Use Case Maps (UCMs), Model Bridging, Rule-based Decision Making, Architecting Requirements, and Weaving Requirements and Architecture Processes), which relate requirements and architectures in 16 dimensions.

As has clearly been reported in the survey, the researchers claimed that there is no approach which provides a comprehensive elucidation for straightforward mapping between these stages. The findings added that although the approaches attempted to reduce the gaps, most of them involve direct human inputs. To further support this argument, it is also mentioned that the tasks involved in
reducing these gaps are mainly based on intuitions and experiences [GEM04]. Interestingly, the survey concluded that if such an association is expected to exist, an attempt to adopt component-based development should be made in early RE phases. Although some of the approaches included such an attempt [GEM06], nevertheless none provided a mechanism to map the requirements to architectures directly.

Among the investigated work in the literature, which will be presented and discussed in Chapter 2, the only work that deals with each requirement is Behaviour Engineering (BE) [MDF11, MD09, Dro03b, Dro03a, GPHSD05, Dro07]. BE defines a tree-like structure called behaviour tree (BT) to represent the requirements. These trees are then integrated to build another design tree, which is then used to derive an architecture and finally, build the system. Although the BE approach deals with requirements incrementally, nonetheless, the construction of the system architecture is performed in a single step. We differentiate our approach from BE in the view of this architecture construction. In this research, both contexts, namely handling the NLR and constructing the system architecture are done incrementally, whilst adhering to the semantics of the X-MAN component model.

The motivation of this research, then, is to reduce these gaps between the requirements and architectures by determining and defining a plausible mechanism for the construction process. By adopting the semantics of a specific component model, inherently all the required features from the model (i.e. encapsulation, reusability and compositionality) will result in side-effects. Hence, transparent transformation will not only promote traceability from the requirements to the architecture, but can also further support the central SE goal, which aims to satisfy user requirements. Moreover, such an approach can also promote reusability as a prominent benefit offer by the CBSE domain.
1.2. PROBLEM DESCRIPTION AND MOTIVATION

In realising the problems in handling the transition between the RE and architecture stages, there is a need to investigate and formulate a new and enhanced solution in order to address the limitations and problems highlighted in the literature. This study seeks to address ways to reduce these gaps through mapping the NLR incrementally using a specific component model, the X-MAN, as specified in [LW06, LW07]. The justification for adopting this model is that the model achieves reusability via compositional semantics. This feature allows composition to be performed incrementally to build a larger composition. In addition, the *encapsulation* feature is also a significant concept that promotes and supports the extraction of component-based elements from natural language requirements. In effect, no dependencies have to be thoroughly checked prior to a composition, which makes this option advantageous among all the rest of the component models available in the literature.

The underlying ideas from the X-MAN component model [LW06, LW07] are based on the following:

(a) Components are pre-existing reusable software units, which can be composed without having to re-write the code from scratch.

(b) Components ideally should be independently produced and used. This will ensure that the components are easily reused.

(c) Components should be able to be copied and instantiated to be fully reusable in different stages of the component life cycle i.e. design, deployment and run-time phases. The component model highlights different aspects of the component design and construction in each phase. These phases will be introduced in Chapter 3.

(d) Components should be able to be composed into composite components,
which in turn can further be composed with another atomic or composite component in a systematic manner.

By adopting the X-MAN component model, inherently the features embedded in the model can be utilised towards achieving the aim to deal with the gap between requirements and architectures.

1.3 Research Objectives

The central aim of this study is to establish the means of incremental mapping of NLR into architectural constructs while gradually constructing the component-based system. The idea proposed in this thesis is to analyse the NLR using textual analysis, extract the relevant keywords and map the extracted elements into an architecture, which conforms to the component model’s semantics. The challenges in this work will mainly be to address the following research objectives:

(1) To formulate and define a systematic approach to incrementally construct component-based systems from raw requirements. By having the underlying component model, inherently, the benefits of the model can be embraced. In addition, any pre-existing components can also be considered during this early stage, thus reducing the workload and effort for the system under construction.

(2) To investigate and define mechanisms to support the transition from requirements to architectures. It is also crucial to investigate ways in which the requirements can be adequately mapped into an architecture in most cases. These mechanisms should therefore be defined to capture all the necessary elements in both RE and CBSE domains.
(3) To map the requirements to architecture incrementally, without breaking the notion of encapsulation and composition of the component model. It is important that the formulated approach should be performed incrementally as the requirements are normally stated in lengthy documents. Hence, by proving that the proposed approach can work incrementally, it will at the same time reduce complexity when dealing in a small-scale requirement at a single point of time.

(4) To evaluate the proposed approach. In order to validate the proof-of-concept applied in the proposed approach, the test cases are executed based on the final system, a preliminary empirical validation is reported, reflections based on the execution of case studies and a comparison with related work are provided.

In essence, the particular focus of this study is to apply the defined approach to a set of requirements stated in natural language, in contrast to any other requirements representation (e.g. modelling artefact). The main justifications for this purpose are that this reflects the nature of real-world scenarios, and such requirements are easier to understand by humans, although the nature of the natural language representation might lead to classical problems in RE. It is worth mentioning here that the aim of this study is not to overcome the classical problems in RE. Nonetheless, it is an irrefutably challenging research opportunity with regard to the intention of reducing the gaps between these two stages.

In regards to traceability, as mentioned in the previous section, having a way to map the requirements to the architecture directly will accordingly contribute towards handling the traceability issue. This is because, if the requirements are mapped one-by-one incrementally, the relationships could definitely be maintained and hence the traceability issue can be appropriately handled.
1.4 The Incremental Approach

This research attempts to derive a means of constructing component-based systems directly from requirements using a mechanism, that is the *incremental composition*. This means is supported by the adoption of the X-MAN component model. The key elements of the X-MAN model are computations, control and data. The computations and control are encapsulated in separate entities in the model, namely computations are encapsulated as data transformations (or provided services) of components, whilst controls are encapsulated as composition connectors.

A component model should help us to design a component-based system with minimal or no coupling between the components, whilst maintaining maximal cohesion within components. This obviously results in less coupling and higher cohesion design, compared to procedural or object-oriented systems. It also means that it should be easier to identify elements of a component-based system individually and separately from the requirements, so much so that we can directly map requirements to elements of a component-based system; that is we can go from requirements directly to system design.

As an overview, this approach formulates a five-step process, starting from (1) extracting elements of component-based systems from NLR, namely computations, control and data; (2) mapping of the extracted elements into the chosen component model constructs; (3) creating a partial architecture; (4) composing the resulting partial architecture into a system architecture and finally, (5) finalising the architecture. Once the final architecture is derived, the system can be executed and tested against a set of test cases. This process is adopted from general testing technique in conventional software development, as it is relevant to be applied. Nonetheless, the main reason to validate the produced system is
1.5. RESEARCH CONTRIBUTION

to test that it satisfies the original requirements.

Based on these processes, a conceptual model of the overall approach can be abstracted, as shown in Fig. 1.1. This model depicts the entire approach, which highlights the relevant aspects of the processes involved. From NLR, component-based elements are discovered using a textual analysis technique. The outcomes of this process are the extracted elements which can be mapped to the component model’s constructs. These constructs help to construct component-based systems using incremental composition.

By adopting a component model that provides an incremental composition mechanism, it allows us to (1) map the extracted keywords directly to their corresponding executable architectural constructs as systems architecture; (2) deal with any number of requirements; and therefore it should scale up to arbitrarily large requirement documents.

1.5 Research Contribution

In this thesis, the research contributes to knowledge as listed below:
1. The research advocates a category of mapping approaches based on:

(a) how the requirements are handled either dealing with the requirements one-by-one or dealing with them as a whole.

(b) how the system is constructed either incrementally or in a single step.

2. The resulting approach defines a new way to incrementally construct component-based systems from raw requirements. The approach has the following features:

(a) It builds upon the formulation of a five-step process starting from extracting elements of component-based systems from NLR, mapping of the extracted elements into the chosen component model constructs, creating a partial architecture, composing the partial architecture into a system architecture and, finally, finalising the architecture.

(b) It defines and justifies the heuristic rules for the elements extraction process.

(c) It defines and justifies the design decisions for the mapping process.

(d) It defines and justifies the design decisions for the incremental composition.

(e) It defines and justifies the design decisions for the architecture refactoring.

(f) It provides a toolset to support the elements extraction process (the Extractor tool).

3. Validation of the approach based on:

(a) feasibility of the incremental approach by means of the provided case studies.
1.6 **SCOPE AND LIMITATIONS**

(b) preliminary empirical validation of the incremental approach.

(c) analysis based on the execution of the provided case studies.

(d) analysis and comparison of related work.

### 1.6 Scope and Limitations

The X-MAN component model is a behavioural model. As this research adopts the model as the underlying framework, the extraction process depends on the elements defined in the model. Hence this research is confined to accommodating functional requirements only.

The fundamental idea that supports the extraction of component-based constructs only relies on its syntactic analysis, and thus excludes this work from claiming beyond this limit. However, some explicit words that can be used semantically for control identification have been discovered. Apart from this, an analyst has to manually select the relevant concepts from the suggested extracted elements.

It has also been demonstrated that the identified computations, control and data serve to construct the complete component-based systems. Although this approach is basically heuristic, and requires human guidance and decision making, we believe this is possible because the underlying component model provides a way to realise such an approach. In addition, the steps and rules that are followed have been adequately defined for the sake of consistency.

### 1.7 Thesis Structure

This thesis is conceptually organised into ten chapters in total, beginning with this *Introduction and Background* chapter. The remaining parts of the thesis
have been structured as follows:

- **Chapter 2: Existing Approaches to Constructing Systems from NLR** begins by laying out the past research efforts in bridging the gaps between requirements and architectures (systems). The chapter examines and analyses the significant results in reducing the gaps between the two stages and looks at the methods used in each approach. In the end, a category of approaches is provided in order to highlight the intended research problem to be solved.

- **Chapter 3: The X-MAN Component Model** presents the foundation of the X-MAN component model. Since the model will be adopted as the underlying notion of the approach used to motivate the extraction and mapping process, this chapter provides sufficient knowledge of the component model.

- **Chapter 4: Incremental Composition** discusses, proposes and justifies a set of heuristic rules for the incremental composition (IC) mechanism to be applied in the approach. Without this IC mechanism, the incremental approach cannot be fully achieved.

- **Chapter 5: Extracting Elements of Component-based Systems from NLR** provides in-depth discussion on the definition of the element extraction process and the proposed element extraction heuristics using a textual analysis technique. For each heuristic, examples are provided and analysed.

- **Chapter 6: Mapping from Requirements to the Architectural Elements** defines and elaborates the mapping of the extracted elements into elements in component-based systems based on the X-MAN semantics. The outcome of this process is a set of mapping elements which are then modelled into partial architectures.
1.7. THESIS STRUCTURE

- **Chapter 7: Architecture Refactoring** presents, proposes and justifies a set of architecture refactoring rules to be used in the mapping process. By adhering to these rules, behaviour preservation property can be guaranteed rather than arbitrarily making changes to the structure of the architecture.

- **Chapter 8: Defining the Approach** defines the holistic mapping approach from requirements to architecture using the incremental composition. This chapter sets out all algorithms and the required processes in each step during each increment. A simple yet complete example is used to demonstrate the entire approach.

- **Chapter 9: Validation and Discussion** demonstrates the evaluation of the defined approach using an empirical validation, case studies and also comparison with related work. An analysis of insights extracted based on the execution of case studies is also included.

- **Chapter 10: Conclusion and Future Work** concludes the thesis by discussing the overall contribution of the research to the SE research area. In addition, this chapter also relates the contribution to the research objectives. Ideas for future work directions are also included.

In addition, the appendices contain (1) background to the textual analysis technique and (2) case studies adopted in discussing this thesis.

Fig. 1.2 depicts the relationships between all the chapters in this thesis. Starting with the first chapter, and continues with Chapters 2 and 3. The following Chapters 4, 5, 6 and 7 are needed to understand the overall approach defined in Chapter 8. The subsequent chapter, Chapter 9, provides the evaluation and discussion of the overall approach, whilst the final chapter concludes the thesis.
1.8 Summary

This chapter has set out the introduction and background of the research. In addition, the research objectives, contributions, scope and limitations have been presented. In order to gain a further understanding of the existing work in building architecture (systems) from requirements, the subsequent chapter will examine them and provide a synthesis based on the analysis. Each of the approaches and their adopted mechanisms will be analysed and discussed.
Chapter 2

Existing Approaches to Constructing Systems from Natural Language Requirements

This chapter presents the analyses and reviews of the existing approaches that claim to be able to map requirements to system architecture. Based on this analysis, a category of mapping approaches is defined. Each of these approaches will be elaborated based on the defined category. Towards the end of this chapter, the research problem being investigated in this work and its significance will be addressed.

2.1 Introduction

System development generally starts with RE process. The outcomes of this process are typically regarded as input to the succeeding processes. In these processes, intermediate requirements models (e.g. object-oriented systems analysis and design (e.g. use case, class, state chart diagrams), structured systems analysis
and design (e.g. context, data flow diagrams)), are derived. These intermediate requirements models are manually\textsuperscript{1} constructed as a result of some degree of the abstraction process. Such abstraction models, at best, only approximate the raw requirements. Therefore, the same is true of the requirements specification that results from the RE process.

Such a scenario can generally be visualised in Fig. 2.1. Based on an input containing a set of raw requirements, a systems analyst abstracts the required information and models it according to any appropriate models in the light of the chosen software development methodology.

For instance, in object-oriented software development, an analyst deciphers a set of requirements and represents the abstracted information in a series of diagrams (e.g. use case, class, sequence diagram etc.) according to the Unified Modelling Language (UML) notation.\textsuperscript{2} This whole process is entirely dependent on the expertise of the systems analysts and the domain experts involved. Eventually, the software programmer implements the system based on these abstracted diagrams.

During the modelling process, crucial information that influences the design decisions might have been overlooked or disregarded. In practice, the modelling process may involve more than one systems analyst, not to mention a number of domain experts. The complexity of this process is further increased by the number of diagrams involved. While producing and maintaining the evolution of this set of diagrams, traceabilities between each modelling element and the original requirements have to be fully addressed. These scenarios are part of the common

---

\textsuperscript{1}Manual here means derived from scratch or semi-automated.

\textsuperscript{2}UML is the \textit{de-facto} standard of object-oriented modelling notation.
2.2. ANALYSIS OF THE EXISTING APPROACHES

issues in RE processes and they are considered sufficient to demonstrate the complexity of the processes. Regardless of these issues, however, it is important to ensure that crucial information is not lost during the early stages as it is the basis of information for design decisions in the later stages.

This raises the questions of (1) how and to what degree of accuracy can these abstraction processes fully elicit, capture and satisfy the entire set of requirements; (2) how the risk of information lost during such an abstraction process can be tackled especially when the models are being transformed from one another in each phase of the development life cycle, and further being handled by different stakeholders in the development team; and (3) how these processes can address scalability when dealing with many stages of transformations while having to keep abreast of the relationships among the software artefacts.

2.2 Analysis of the Existing Approaches

A considerable number of existing approaches that deal with handling requirements with regards to architecture have been investigated. Through an analysis of these approaches, a category of mapping approaches is defined. Fig. 2.2 depicts the idea of the defined category. The category is established based on (1) how these approaches deal with requirements; and (2) how these approaches model information from the requirements into systems architecture.

For the first criterion, we look for approaches that deal with each single requirement or deal with an entire set of requirements. As presented in Section 2.1, an approach that can systematically deal with each single requirement can at least reduce the possibility of disregarding any useful information stated in the requirement. The main reason is that each requirement will definitely be handled at some point. The current state of requirements elicitation and analysis
processes are solely subjected to the abstraction made based on the judgment and expertise of the systems analyst. This abstraction process may also run the risk of losing useful details from the requirements. In order to overcome such an issue, various types of traceability techniques are introduced in the literature [Got95, RJ01, ACDL99, EEM95, HDS06, VCW+05]. Accordingly, more cost and effort are required to produce traceability links and to maintain them.

For the second criterion, we differentiate between approaches that build systems architecture in a single step and approaches that build systems in incremental steps. Building systems architecture in a single step is an approach that analyses the relevant information (derived from the mapping process, see Fig. 2.2) and can directly produce a system architecture. On the contrary, the other approaches build the system architecture incrementally or by refinement processes. Through the refinement process, abstracted analysis and design models will be refined into detailed design. An essential strategy in the object-oriented analysis and structured analysis is the decomposition of a problem into smaller and manageable units. This strategy is also known as the divide-and-conquer technique. This technique supports such refinement process towards achieving the detailed design. On the other hand, an incremental approach allows us to deal with any number of requirements, and therefore it should scale up to arbitrarily large requirements documents.

These mapping approaches will be discussed as based on the defined category shown in Fig. 2.2. The category column represents the following: (1) approaches that deal with all requirements and build the system architecture in a single step; (2) approaches that deal with all requirements and build the system architecture incrementally (or by refinement); and (3) approaches that deal with a single requirement at each incremental step and build the system architecture in a single step.
2.2. ANALYSIS OF THE EXISTING APPROACHES

It is worth noting that the details of these approaches are not included in this section; however, they will be adequately discussed in the context of how we defined the categories i.e. how these approaches deal with functional requirements, and how these approaches handle and derive the systems architectures.

It is useful to distinguish between the terms ‘refinement’ and ‘incremental’ used in this category. By refinement, we mean the process of detailing the design specification. In literature, such a refinement process is also known as forward engineering [EM99]. This process is normally applicable in a top-down approach where an abstraction concept is identified, and details are added to the corresponding models. For instance, in a feature-orientation approach, each feature is an abstraction of the generic properties of a system being modelled. This feature needs to be refined to achieve the designated design artefact. On the other hand, by ‘incremental’, we mean the process of adding increments to the architecture or system but at the same time preserving the formerly incremented behaviour without compromising its correctness. A refinement can be, but is not necessarily, incremental. Typically, a refinement process is a top-down activity, while an incremental process supports bottom-up activities.
2.2.1 Category 1: Deal with all requirements and build the system architecture in a single step

In this category, the entire set of requirements are analysed and relevant information based on the corresponding selected domain models is extracted. In these tasks, the expertise of the systems analysts and the domain experts is obviously required. The produced set of domain models will be used as a basis for later software development stages. When all the required information is elicited, analysed and modelled (in addition to the existing domain models from the earlier stages), systems architecture will also be derived in the same manner.

Most of the studied mapping approaches fall into this category. Among the examples are the Architectural Decision Elicitation Framework (ADEF) [LE03], the Analysing Requirements Trade-offs - Scenario Evaluations (ART-SCENE) [ZMP03], the Extended Design Spaces (EDS) [BBGM00] and the Scenario and Meta-Model-Based Approach (SMM) [PBG01].

The following sections include an introduction to each of the approaches in this category, and continuing with an analysis of each of the approaches based on the theme of the defined category.
1. Architectural Decision Elicitation Framework

The Architectural Decision Elicitation Framework (ADEF) [LE03] approach provides not only design guidance in capturing architectural decisions from the requirements using a \textit{rule-based implementation}, but also contains automatic reasoning capability for making architectural decisions and resolving conflicting decisions.

The ADEF approach comprises two main modules, namely (1) \textit{reasoning} and (2) \textit{presentation} modules, as depicted in Fig. 2.4.

The \textit{Reasoning} module encapsulates the decision making knowledge and justifications on the requirements elicitation to be mapped to any relevant architectural decisions. This module contains three sub-modules which are: (1) \textit{Mapping}. This sub-module uses built-in decision trees represented in directed acyclic graphs to provide guidance to the user in manually mapping each requirements specification to one or more architecturally significant properties; (2) \textit{Conversion} During the conversion process, the decision units will be converted to a form which can be interpreted by the analysis of the sub module. These decisions are stored as \textit{facts}; (3) \textit{Analysis}. In analysis, an automated reasoning is provided in order to make the architectural decisions and to resolve conflicting decisions. Finally, the \textit{Presentation} module acts as the presentation layer to the user and also updates any changes made to the preceding results.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ADEF_diagram}
\caption{ADEF [LE03]}
\end{figure}
Analysis. The ADEF approach adopts straightforward transitions by analysing the entire set of requirements, and relates the mapped architectural significant properties into architectural units (e.g. component, connector or bus, system). Clearly, the construction of an architecture of a system is performed in a single step. For this reason, this approach is classified into the first category.

2. Analysing Requirements Trade-offs - Scenario Evaluations

The Analysing Requirements Trade-offs - Scenarios Evaluation (ART-SCENE) approach is designed with the intention of utilising an integration of the existing tools and techniques to relate requirements and architecture [ZMP03]. By integrating the chosen RE techniques and tools, the ART-SCENE approach claims to provide a comprehensive plug-and-play approach in exploring the requirements and architecture trade-offs with different scenarios.

This approach advocates scenario generation, architectural model and agent-based simulation to explore architectural models for various scenarios [ZMP03]. The researchers claimed that scenarios are essential in integrating requirements and architectures.

Scenarios, in this approach, are used to realise requirements and to simulate the choice of architectural design, which are then validated for compliance with the requirements as shown in Fig. 2.5.

Analysis. In the ART-SCENE approach, the whole set of requirements is analysed and modelled into a group of scenarios. While abstracting the requirements into scenarios and modelling them into use cases, important requirements or information
might be lost. This design decision is based on the expert judgment and intuition of the involved systems engineer or the systems analyst. Based on this abstracted information, the engineer or analyst models the system requirements using the \( i^\text{notation} \)\cite{MPG+02}. Architecture is then designed (without refinement) and simulated using the AgentSheets\cite{FH01, RIZ00} tool.

3. Extended Design Spaces

The Extended Design Spaces (EDS) is a semi-formal technique, which aims to achieve reusability in CBSE by providing a set of processes from requirements capturing to configuration \cite{BBGM00}. The authors argue that to foster component reuse, software development processes should be geared to consider reusability aspects during the processes.

The EDS process model starts by mapping requirements to components. See Fig. 2.6. In many of the existing RE approaches, requirements are stated without any consideration of the architectural designs. To some extent, this is the desirable RE feature, because RE is expected to cover the what part of the problem, and not the how part of the solution. However, the drawback of this is that from the reusability aspect, the developers need to manually decide how to bridge these gaps \cite{BBGM00}. In order to take advantage of this situation, the EDS provides guidance in the form of a checklist, which contains suggested types of requirement according to the chosen type of architecture.

Firstly, a DS (Design Spaces) profile is created. The outcome of this is the Requirements DS (RDS) which comprises a list of functional and non-functional
aspects of the system. The Application DS (ADS) captures the properties of the application, while the RDS describes the requirements in the perspective of the run-time platform. The subsequent step is to map the ADS to the RDS. Some of the ADSs are not directly mapped to the RDS. Thus, some transformation rules are required. These rules are derived through the correlation between the two spaces.

Secondly, the DS profile will be used to select the architecture. An architecture, in this context, specifies the design decisions and component compatibility. Components that are built for a specific type of architecture share the same assumptions about the design decisions. There is no doubt that the decision about the most appropriate architecture largely depends on the expertise of the software architect. Nonetheless, by using the correlation results from ADS and RDS, the choice of architecture can be justified.

Thirdly, components are selected, configured and composed. Component selection for reuse purposes is done based on the component profiles. The component selection process will then compare the desired DS profile with the provided DS profile from the repository. If there is no match, then a new component will need to be developed from scratch. The component configuration provides the means to determine the parameter settings of the components which are retrieved from the repository. Complex parameters that cannot be handled using the DS technique will need to be manually resolved (e.g. informal hints).

Finally, when all the desired components have been retrieved and configured, a generator uses the components together with their parameter settings and translates them into implementations [BBGM00].

**Analysis.** The EDS approach deals with the whole set of requirements in an attempt to map functional and non-functional requirements into an architecture.
2.2. **ANALYSIS OF THE EXISTING APPROACHES**

On the other hand, the construction of the architecture is done in a single step once the desired components have been discovered. With this justification, we assign the EDS approach to the first category.

4. **Scenario and Meta-Model-Based Approach**

The Scenario and Meta-Model-Based (SMM) approach defines the gap between requirements and architecture as a *structural* gap [PBG01]. The existence of this gap is claimed because of the large conceptual distance between the terms used in both of the stages. A requirement may be modelled into one or more components, whereas one component may relate to one or more requirements. Based on the claim that scenarios can be used as a means to elicit the inter-relationships between requirements and architecture, in relating both of the stages, the SMM approach adopts scenario integration in these stages. Nonetheless, the main aims of this approach are to establish traceability from requirements to implementation and to support change integration.

Meta-models contain a set of descriptions to capture the modelling elements used in a model-driven environment. In this approach, the meta-models are represented using a UML class diagram. These meta-models can be specified into domain-specific meta-models by adding specialised domain concepts. Altogether, six meta-models have been defined to capture the modelling elements with regard to relating the requirements and architecture. These meta-models are illustrated in Fig. 2.7. During the requirements stage a *requirements meta-model* identifies FRs and NFRs; a *use case meta-model* defines goals, external actors, triggers, pre-conditions, post-conditions, results and use case scenarios;

---

**Fig. 2.7: Scenario Meta Model [PBG01]**
a use case scenario meta-model includes the satisfy use case scenario (i.e. primary and alternative) and exception use case scenario, and scenario steps (i.e. action and communication).

During the architecture stage, architectural meta-model specifies internal components and connectors. An architecture consists of a collection of components and connectors. An architectural configuration meta-model identifies the configurations of the components and connectors. An architectural scenario meta-model represents the dynamic behaviour of the architecture. It is a realisation of some use cases, in which the architecture scenario steps (i.e. action and communication) are defined.

Analysis. The SMM approach deals with the whole set of requirements in order to specify scenarios. These scenarios are used to identify the elements of the defined meta models in order to look for architectural elements. The architecture of the system is derived once all the architectural elements are discovered. Hence, we classify this work in the first category. In addition, we believe that the abstraction process of going from requirements to the scenarios, and the process of identifying the elements of the meta-models are based on the intuition and expert judgment of the analyst (or the designer), instead of relying on the methods that this approach has to offer.

2.2.2 Category 2: Deal with all requirements and build the systems architecture incrementally or through refinement

The approaches in the second category deal with the entire set of requirements, which has the same characteristics as the first category. Nonetheless, in contrast
2.2. ANALYSIS OF THE EXISTING APPROACHES

to the first category, the construction of the architecture is performed by means of refinement or incremental process. In this section, we will elaborate and highlight how these approaches handle requirements, and how the architecture is derived from the requirements.

The ability to derive the architecture incrementally while at the same time ensuring the incremented behaviours are maintained without compromising its correctness has a significant and promising impact. The essence of incremental is the fact that it can address the issue of scalability. When progressing with small increments, the architecture can be built up and eventually the final architecture can be derived. Each increment has already been validated and hence, less or ideally no rework effort has to be invested whenever additions are made.

On the other hand, construction of an architecture by means of refinement can also build the architecture hand-in-hand with the requirements. As stated earlier, refinement does not necessarily support incremental. This in turn reflects that preservation of behaviours in the former refinement steps may not be guaranteed. The examples of approaches that are considered in this category are the Feature Orientation (FO) [LM03], the Twin Peaks (TP) [HJL02], the Component-Bus-System-Property (CBSP) [GEM04, GEM01], the Goal-Oriented Requirements Engineering (GORE) [vL04, vL01, DDMvd97], and the Software Architecture-Oriented Requirement Engineering (SAORE) [Mei00].
1. Feature Orientation

The FO-based approaches intended to develop a family of applications for a specific domain instead of for a single application [LM03, SPR04, KLD02, DK, SRP06, LKCC00, LKL02]. A feature is defined as “a logical unit of behavior that is specified by a set of functional and quality requirements” [Bos00]. The core technique in FO is the analysis of variability and commonality of an application domain (i.e. the feature analysis) [LKCC00, LKL02]. These features are then transformed into feature models (see Fig. 2.9), defined in the Feature Oriented Domain Analysis (FODA) technique and used as an input to a series of refinement processes in order to derive an architecture.

![Feature Model](image)

Fig. 2.9: Feature Model

However, the FODA technique does not provide explicit interaction between features and its corresponding architectural elements [SPR04]. Both the FODA and its modelling technique, namely the Feature-Oriented Requirements Modelling (FORM), remain vague on the matter of mapping feature models to architectural elements [SPR04]. In order to overcome this issue, the feature-oriented mapping process is proposed [LM03]. The process comprises two main stages as shown in Fig. 2.10.

The stages are (1) Feature-Oriented Requirements Modelling (FORM) and (2)
2.2. ANALYSIS OF THE EXISTING APPROACHES

Feature-Oriented Architectural Modelling (FOAM). The aims of the FORM stage are (1) to capture the problem domain requirements and transform them in the context of features and their relationships (i.e. during the elicitation process); (2) to analyse and transform the features into a tree form (i.e. during the organisation and analysis process), (see Fig. 2.9); and (3) to refine the features into a detailed set of functional requirements (i.e. during feature refinement).

Moreover, in the second stage, namely the FOAM stage, the outcome of the FORM stage serves as an input to this stage in order to derive an architecture. A functional feature can be mapped to a subsystem or a component. In FOAM, mapping is addressed in three layers, which are conceptual, logical and deployment architecture as illustrated in Fig. 2.10. In the conceptual architecture, only functional features are considered. By taking implementation and non-functional features into consideration, these features are detailed for the logical architecture. Finally, the deployment architecture represents how the features are distributed and communicated among computational nodes in the deployment environment.
Analysis. These FO approaches deal with an entire set of requirements and abstract them as features. This abstraction of features together with their relationships is represented as a feature model. Moreover, the construction of an architecture is refined into a series of conceptual, logical and deployment architectures. Clearly the FO-based approaches work on the basis of dealing with the entire set of requirements and transforming the abstracted information into an architecture through the refinement process. For this reason, we assign the FO approach into the second category.

2. Twin Peaks

The Twin Peaks (TP) model adopted the Problem Frames (PF) [Jac95] approach. Thus, some foundations of the PF approach will be briefly introduced. The PF approach attempts to decompose problems into a collection of interacting sub-problems and to manage the separation of the real-world problems into the problem and solution spaces [Jac95]. By having such a separation, a developer can focus on the problem structure while attempting to solve the real-world problem rather than trying to find a solution to the problem. Nevertheless, coming up with the problem domain without considering the solution domain is, if not impossible, rather a difficult task.

In order to reduce the gaps between problem and solution spaces, and to allow requirements and design opportunities to be explored, current software development should exploit the relationships that exist between these spaces [Bas01]. This approach defines frames to a particular problem in which each frame has an associated frame concern, and does not address the concerns of the software architects [HJL+02]. This frame concern identifies the correctness argument of the frame. By excluding these concerns, the development process is less likely to be shortened [HJL+02]. Hence, the PF approach is extended to consider the
solution spaces as a part of the problem domain [HJL+02].

In relating both of the spaces, the iterative nature in software development is also taken into consideration, from the general to a more detailed design solution. During this process, some architectural support decisions are used in the problem space. In order to address the issue, the TP model highlights the equal importance of requirements and architectures by separating the problem space and the specification from the solution space [Bas01, HJL+02].

The specification comprises detailed requirements and design specifications which are produced by means of refinement. Here, the refinement is confined to the process of detailing the specification, hand-in-hand with the requirements. This concept is illustrated in Fig. 2.11. The model mainly addresses the problems dealing with evolving user requirements, applying and managing COTS software, and handling rapid changes [Bas01]. This approach adopts a causal logic event model [MHCM96] to express the event notation, and also applies an informal imperative pseudo code to describe behaviours.

The first TP mapping step is to decompose the problem into independent sub-problems i.e. not having any relationship with any other problem or sub-problems. Each sub-problem will be represented according to the chosen notation. Once all of the sub-problems have been modelled into the commanded behaviour frame notation, the requirements will be represented in the form of causal logic event model expressions.
Analysis. The TP approach deals with the entire set of requirements, and refines the specification by attempting to reduce gaps between the problem and solution spaces. Approaches in the second category handle requirements by processing the whole set of requirements. The fact that the requirements specification is refined in TP indicates that the mapping process itself addresses such a refinement process. Nonetheless, the requirement must be analysed as a whole, rather than dealing with each requirement at a single point in time. The second consideration is regarding the construction of the architecture. In TP, architectures are derived through the refinement process, hand-in-hand with the refinement of requirements.

3. Component-Bus-System-Property

The Component-Bus-System-Property (CBSP) is defined as “a lightweight approach which provides a systematic way of reconciling requirements and architecture using intermediate models” [GEM04, GEM01]. This generic approach adopts the Twin Peaks Model [Bas01, HJL+02] to exhibit the iterative process between requirements and architecture. In addition to this, an intermediate model is introduced between these stages, as shown in Fig. 2.12.

This model helps to decide on a suitable architectural style to be adopted as a basis of transforming the draft architecture to an actual software architecture implementation [GEM04].

The quintessential concept in CBSP is that each requirement will implicitly or explicitly relate to the software architecture elements. The CBSP defines dimensions which consist of a set of architectural elements (i.e. components, connectors

Fig. 2.12: CBSP Conceptual Framework [GEM04]
2.2. ANALYSIS OF THE EXISTING APPROACHES

(buses), topology and their properties). These dimensions are used to classify and refine the requirements in order to capture architectural-related issues.

As depicted in Fig. 2.13, the mapping process begins with requirements selection. The selection is a voting-based process where requirements with low priorities will be eliminated. This voting process is conducted by stakeholders of the software project team. Accordingly, the main criteria for voting are the importance and feasibility of the requirements. The second step is to classify the requirements into architectural constructs. This step is performed by experts. In order to classify the constructs, the CBSP dimensions are introduced. The CBSP approach specifies six dimensions to be applied to the basic architectural constructs. These dimensions represent elements that imply (1) processing or data components; (2) a connector or bus; (3) features of a subset of the components and connectors; (4) NFR aspects of a component; (5) NFR aspects of a

Fig. 2.13: CBSP Process [GEM04]
connector; and (6) system (sub-system) properties. These dimensions are used to refine the requirements in natural language into CBSP model elements. The experts will then examine each of the requirements and vote on the relevance based on the CBSP dimensions.

The third step is to identify and resolve mismatch classification issues. This is where any conflicting perception during the classification in the previous steps will be handled. Again, a voting process is adopted. If there is still no consensus among the experts, a discussion is needed so as to avoid the conflicts and to achieve the final decision.

In the fourth step, each of the requirements will be refined based on overlapping CBSP properties and concerns. Finally, at this step it is assumed that all the conflicts have been resolved. Based on the CBSP model elements a proto-architecture will be derived. This is also the point where architectural styles (e.g. client-server, event-based, layered and pipe-and-filter architecture) [Fie04] will be used to achieve the desired system qualities. This process is heuristically performed, with the justification that there are also possibilities that more than one architectural style is needed or preferable. By using the elements in the selected architectural style(s), the CBSP model elements will be transformed into components, connectors, configurations and data.

**Analysis.** The CBSP mapping process is an iterative process that is used to refine the draft architecture model derived from a given set of requirements. In the solution structure, the architecture is matched with the existing architectural styles. This architecture is refined iteratively until the completion of the model. The CBSP is considered under this category because although each requirement is voted, and the CBSP elements are identified, the architecture is not created based on the behaviours that directly originated from requirements. Instead, the chosen architectural style is refined, together with the original requirements as
2.2. **ANALYSIS OF THE EXISTING APPROACHES**

illustrated by the spiral dotted line in Fig. 2.12.

4. **Goal-Oriented Requirements Engineering**

A *goal* is “a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some of the agents that form the system” [vL03]. These goals cover both functional and non-functional goals (NFG). In brief, this approach starts by providing a process to produce software specification based on a set of requirements. From the functional specification, an initial abstract architecture is drafted. Following this, the architecture is refined to meet the domain-specific architectural constraints.

The Goal Oriented Requirements Engineering (GORE) approach adopts Knowledge Acquisition in Automated Specification (KAOS) methodology [vL04, vL01, DDMvd97] in addressing the architectural design based on the requirements. The KAOS methodology specifies an *ontology* for capturing requirements [DDMvd97]. The relevant part of this approach is the architecture derivation process, which happens after the requirements are modelled, specified and analysed. Such a derivation process (1) provides a systematic guidance to software architects; (2) is incremental; (3) leads to an architecture that satisfies the functional and NFR; and (4) allows different architectural views (e.g. performance view, security view, etc.) [vL03].

The GORE process can be abstractly viewed according to the functional and NFG derivation processes. As shown in Fig. 2.14, for the functional goals, all the relevant documents are referred to in order to identify system goals. Based on these system goals, software requirements are identified. During the following step, the whole set of requirements is analysed and software specifications are further derived. Finally, the abstract data flow architectures are derived, based on the software specification.
Analysis. In the GORE approach, the architecture refinement process is specifically required to include the NFG. In the context of functional properties or behavioural properties, the refinement process is not required. However, without the NFG, the derivation process might not be complete. For this reason, this approach is classified in the second category.

5. Software Architecture-Oriented Requirement Engineering

The Software Architecture-Oriented Requirement Engineering (SAORE) approach attempts to relate the requirements and architecture stages[Mei00]. In essence, the approach views the architecture model as the reference model in the software development process. The main idea of this approach is to introduce elements of software architecture (SA) in requirements analysis and specification. The SAORE approach identifies two main constituents: components and connectors. A component is defined as an element which encapsulates a set of coherent functionalities, which is expected to perform the desired computational units. A connector, on the other hand, defines the interaction between components. The connector is treated as a first-class entity not only in the problem space, but also in the solution space. Examples of the connectors are procedure calls, file Input/Output (I/O), remote procedure calls (RMC) and client-server.

The SAORE process [Mei00] comprises a series of iterative activities, as illustrated in Fig. 2.15. The starting point of the process is to determine the boundary of problem spaces i.e. the scope of the system to be built. Following this, stakeholders of the system and their roles are identified. The information is
2.2. ANALYSIS OF THE EXISTING APPROACHES

useful to determine the external relationships of the system and also to partition
the system into sub-systems or components. In the next step, a top level system
model will be produced using a Use Case model. The challenging step here is
how to identify the components and connectors from the elicited information.

Analysis. The SAORE approach deals with the entire set of requirements and
analyses them. The components are extracted from use cases or any element that
contributes system behaviours, whilst the connectors encapsulate relationships
between components. Consequently, the requirement models are derived through
a series of refinement steps and this is why the approach is addressed as in the
second category.

2.2.3 Category 3: Deal with a single requirement at each
step and build the system architecture in a single step

In contrast to dealing with the entire set of requirements, the approach in this cat-
egory deals with each requirement and processes the requirements incrementally.
As a result, the architecture is constructed once the complete design specification
CHAPTER 2. EXISTING APPROACHES TO CONST. SYSTEM

is derived. This process is therefore done in a single step.

<table>
<thead>
<tr>
<th>Cat. Requirements</th>
<th>Architecture</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Mapping Process</td>
<td>BE</td>
</tr>
</tbody>
</table>

Fig. 2.16: Category 3

To the best of our knowledge, as shown in Fig. 2.16, the only work that fits into this category is the Behaviour Engineering (BE) approach.

1. Behaviour Engineering

Behaviour Engineering (BE) establishes an approach for building large-scale systems from natural language requirements. The principle of the BE approach lies in its concept of building design out of requirements instead of satisfying the requirement [Dro03a]. It defines the Behaviour Modelling Process (BMP) which comprises a set of processes to support Behaviour Modelling Language (BML). The BML consists of a series of behavioural diagrams including Behaviour Trees (BT), Integrated (IBT), Model (MBT) and Design (DBT) [Mye10, MDF11, MD09, Dro03b, Dro03a, GPHSD05, Dro07]. A BT translates individual (raw) requirements into behaviour trees.

A BT is “a tree-like graph that represents the behaviour of a set of entities which realise or change states, make decisions, respond to or cause events, and interact by exchanging information and/or passing control” [Dro03a].

Behaviour trees for all the individual requirements (called requirements behaviour trees (RBTs)) are merged into an IBT that describes the required behaviours of the whole system. Fig. 2.17 shows examples of the RBTs for two requirements R1 and R2 of an ATM Example and their merged IBT. The IBT is
2.2. ANALYSIS OF THE EXISTING APPROACHES

Fig. 2.17: Behaviour Trees

used to integrate all the RBTs. The following diagram, that is the MBT, is used to discover and fix defects on issues such as inconsistencies, ambiguities, etc. As a result of the defect correction, a design behaviour tree (DBT) is produced.

The DBT is again refined using the provided design decisions. This refinement process clarifies the system’s boundaries. From the DBT of the whole system, a component diagram [MDF11, Mye10] is derived, together with the behaviours of individual components.

Analysis. According to the approaches category, BE clearly deals with each single requirement by producing these RBTs. Nevertheless, when the complete DBT is produced through refinement, only then is the component diagram constructed. This process is performed in a single step. In such a case, iteration or refinement from design to the system architecture diagram is not required. For these justifications, the BE is considered as in the third category.
2.2.4 Category 4: Deal with a single requirement and build the system architecture in each incremental step

Based on the analysis so far, there is a gap in the way that we can handle requirements and at the same time utilise the benefit of incremental approach towards building system architecture. In this new category, each requirement is analysed one-by-one and the result of the analysis is used to aid in creating an architecture incrementally (in contrast to in a single step or through a series of refinement processes). To the best of our knowledge, none of the existing approaches in the literature offer such a strategy. As a consequence of dealing with each require-

![Fig. 2.18: Category 4](image)

As the BE work is closely related to this research, a comparison will be provided in Chapter 9.
2.2. ANALYSIS OF THE EXISTING APPROACHES

2.2.5 The Proposed Approach

The proposed approach is to extract elements of component-based systems from raw requirements, map them directly into the elements of the desired system, and thereby construct the system. The aim of this approach is to maximise the chances of achieving a better match between the final system and the raw requirements. We believe that by having support from a suitable component model, we can achieve such an aim.

It is necessary to consider if we deal directly with each requirement, analyse and model them without having to deal with the whole set of requirements as conceptualised in Fig. 2.18. Such a strategy assists in relatively reducing the complexity that the systems analysts or the domain experts need to handle. Nonetheless, we are not arguing that dealing with each requirement can directly lead to simplicity. The challenge with this kind of approach is how we can effectively compose the mapped architecture for each requirement in each increment.

Reflecting on the category of approaches to construct architecture from requirements, the significant benefit of having these two strategies in an approach is mainly to address scalability. In principle, we may handle a larger set of requirements when we deal with a single requirement at a time and incrementally build up the system architecture.

The problem with the existing approaches in the literature in order to solve the gap between problem and solution, be it using structured or object-oriented approach, is that they provide insufficient emphasis on software architecture elements [LM03]. As a result of this, transformation from requirements to design is arbitrarily performed based on the intuition or judgment of the involved software engineers.

These gaps between requirements and architecture exists because of the large
conceptual distance between the terms used in both stages [PBG01]. In many of the existing RE approaches, requirements are stated short of consideration of the architectural designs. To some extent, this is the desirable RE feature because RE is expected to treat the what part of the problem, and not the how part of the solution. However, the drawback of this feature is that from the reusability aspect, the developers need to manually decide how to bridge these gaps [BBGM00].

In addition to scalability, by adopting a component-based approach, we could reuse pre-existing components from component repositories; in contrast to building new parts of the system for each software development project. This factor leads to reducing effort, time and cost as compared to building new parts of the system from scratch. We argue that by having an underlying component model that supports incremental composition, such an objective can be accomplished. In order to realise the objective, an approach founded on a specific component model should provide a plausible transition from requirements to system architecture for a component-based system.

2.3 Summary

In this chapter, the relevant approaches in the literature have been investigated and analysed. These approaches have been discussed in the light of a defined category of approaches in order to highlight the intended research problem. In order to gain a greater understanding of the application of a specific component model to this work, the following chapter provides the substance of the chosen model.
Chapter 3

The X-MAN Component Model

In order to adopt the component model, it is crucial to gain a firm understanding of the fundamental concepts and terms used in the X-MAN component model. Hence, the main aim of this chapter is to provide these important elements to support the understanding of the component model. The following subsections introduce the basic concepts with respect to the component model. These concepts are widely adopted in the component model, and also relevant to include here, as the notion of the component model will be applied in the following chapters.

3.1 Component Models

A component model [LW07] defines components and their composition. A good component model should enable us to define component-based systems with minimal coupling between components and maximal cohesion within individual components. Coupling results from external dependencies, direct or indirect, between components, and is induced by the composition mechanisms of a component model. For example, in architecture description languages [MT00], components are architectural units with ports, and are composed through a port connection.
Such connections induce external dependencies between components.

High cohesion in a component means that altogether, the sub-components of the component do not have many external dependencies. It is therefore also a consequence of the composition mechanisms of a component model. A composition mechanism that allows the definition of a composition of components will increase cohesion if it allows the composite to have fewer external dependencies than its sub-components. For example, some architecture description languages (ADL), e.g. UML2.0 [Boc04] allow composite architectural units to have arbitrary numbers of ports, and could therefore increase cohesion. A component model that specifies systems with no coupling at all, has been defined. This way, the complete absence of coupling also means the components have maximum cohesion.

In order to adopt a specific component model, namely the X-MAN component model [LLV07, LLW06] as the implicit essential of the proposed approach, it is crucial to get a firm understanding of the fundamental concepts and terms used. This chapter’s main aim is to provide those fundamental concepts of the component model. These concepts are relevant to be included here as the notion of the component model will be applied to the subsequent chapters. The chapter starts by introducing what is a component model and follows by the component life cycle. The subsequent subsections introduce the fundamental concepts defined in the component model and then elaborate the strengths of the component model. At the end, a simple example is included to illustrate the generic usage of the component model.
3.2. WHAT IS A COMPONENT MODEL?

3.2 What is a Component Model?

A component model advocates the “set of component types, their interfaces, and additionally, a specification of the allowable patterns of interaction among component type” [BBB+00]. In general, a component model defines how the components are composed or assembled, their interfaces and the communication among them. A component model should define the semantics, syntax and software component composition [LW06].

3.2.1 Semantics

A component is an independent piece of software that provides functionalities. The term independent here indicates that a component does not rely on any other component to perform any of its functionalities. The semantics of a component merely comprise the questions of how the components represent basic units of the software and how they support composition to build the whole system. These semantics should also define the aspects of (1) identification of the components; (2) definition and construction of an interface; (3) realisation of a component.

3.2.2 Syntax

Syntax describes the way components are represented and built in a specific programming language. It can be represented in the form of a programming language, a component definition language or specification language, e.g. ADL. By using ADL, which is generic, the representation can be implemented in many programming languages.
3.2.3 Composition

Composition refers to the mechanism used to assemble the software components into a larger block or the whole system. In other words, two or more components can be composed together and the outcome of it is also expected to be compositional. The mechanisms for component composition can be either containment, extension, connection or coordination [LR10]. The X-MAN composes components using coordination via exogenous connectors.

3.3 Component Life Cycle

This model defines a component life cycle in three phases, these being design, deployment and run-time. The life cycle can be illustrated in Fig. 3.1 and each of the cycles will be explained based on this figure.

![Component Life Cycle Diagram]

Fig. 3.1: Component Life Cycle

3.3.1 Design Phase

Components can be composed in either the design or deployment phases. Composition in the design phase ideally conforms to the following [Lin07]:

---

**Chapter 3. The X-Man Component Model**

CHAPTER 3. THE X-MAN COMPONENT MODEL
3.3. COMPONENT LIFE CYCLE

1. Components are existing reusable software units;

2. Components are independent software units which can be built by independent component developers;

3. Components can be copied in the design phase and can be instantiated during the run-time phase;

4. Components should be able to be composed into a larger composite using composition operators.

During the design phase, the objective to be achieved is to maximise reuse. In this component model, components are built in the design phase to be further reused in both design and deployment phases. Components are designed, built and then stored in a repository. During these tasks, components are created in source code, which are not executable before compilation and deployment stage. Subsequently, the designed components are catalogued and stored for further purposes. As exemplified in Fig. 3.1, component A is created and stored in the repository, which comprises existing B, C and D components. Assuming that we want to create a composite component (i.e. component BC), the corresponding B and C components will be retrieved from the repository. Consequently, these components can be composed as the BC component using the design phase composition connectors. As the intention of this phase is to maximise reuse, the resulting composite component can also be stored in the repository. Composition during the design phase is generic, and not meant for the design of any specific application.
3.3.2 Deployment Phase

In the deployment phase, components are retrieved from the component repository and compiled into binary codes. During this phase, composition operators compose these binary units into an executable system which can be instantiated in the run-time phase [Eli08]. During deployment, components are retrieved from the repositories and are composed into a targeted environment. At this stage, components are retrieved from the repositories and compiled into binary codes. Binaries of components are now in the form of an executable system. In Fig. 3.1, A, B, D, and BC components are retrieved from the repository, then compiled to assemble them into binary codes. The outcome of this phase is an executable system in binary codes. In contrast to the composition during the design phase, composition during the deployment phase is performed for a specific application. Thus, the result of composition during the deployment stage is not stored in the repository. It is unlikely that a composition that is built for a specific application is useful for reuse.

3.3.3 Run-Time Phase

During this phase, there is no new composition involved. The whole executable system will be instantiated with a set of specific data and executed in a specific Run-Time Environment (RTE). As shown in Fig. 3.1, instances of A, B, D, and BC components are created. As a result, the outcome can be executed in a targeted environment.
3.4 The X-MAN Component Model

As this work adopts the X-MAN component model [LLV07, LLW06], the background of the component model is presented. The fundamental elements of X-MAN are computations, control (and data). Each of the computation and control elements can be explicitly identified and separated in the model. In brief, computations are encapsulated in components, while controls are encapsulated in connectors or adapters. For data, only those that are relevant for checking constraints and storing values are considered.

Based on the findings of a survey [LW07], the component model defines the use of exogenous connectors to support communication between components. The exogenous connectors initiate method calls in the components and encapsulate the control between the components. By defining these connectors, components respond to their connectors only, instead of interacting with other components. A specific subsection discussing the strengths of the X-MAN component model is provided in Section 3.6.

In the component model [LVW05, LOW06, LLW06], computation and control are defined separately.\(^1\) A computation is defined and encapsulated in components whilst control is defined and encapsulated in composition connectors. Components do not call one another; instead, composition connectors define and coordinate all the control among components.

Fig. 3.2 shows the basic elements of the component model. An atomic component (see Fig. 3.2(a)) contains a computation unit (U), and an invocation connector (IU). A computation unit provides methods or functions that can be invoked via the invocation connector. When invoked, the computation unit performs the computation entirely within itself, and is thus encapsulated (i.e. 'enclosed in a

\(^1\)For simplicity, it is assumed that data flows with control.
capsule'). As a result, an atomic component encapsulates computation and has only a provided interface (denoted by a lollipop) and no required interface. Parameters for invocation are passed from composition connectors via the invocation connector.

Components are composed by composition connectors (Fig. 3.2(b)). A composition connector receives control and returns control; it also defines a control structure that determines the control flow between receiving and returning control. A composition connector thus encapsulates control. The composition connectors define the generic control structures: sequencing and branching. For sequencing, the model defines the Sequencer (SEQ) and Pipe (PIPE) connectors.\(^2\) In addition, the model defines the Selector (SEL) connector (see Fig. 3.3) for branching execution.

In a composition (see Fig. 3.2(c)), the composition connector coordinates the control flow between the sub-components. This is illustrated in Fig. 3.4 for the SEQ composition connector.

\(^2\)A PIPE passes results (data) on, whereas a SEQ does not.
3.4. THE X-MAN COMPONENT MODEL

A simple example of a composition is the bank system (see Fig. 3.3) that composes an ATM with a bank consortium (BC) by using a PIPE connector. The customer details and requests are passed to the ATM, which validates them and then passes them on to the BC.

Looping is not a composition connector, because the control for a loop only applies to a single component. It is therefore an adaptor. A loop at the top-most level of a system can be infinite, but elsewhere it must be finite, in order that compositionality is preserved throughout.

The result of a composition is another component with a provided interface (see Fig. 3.2(c) and the bank system in Fig. 3.3). This means that composition in the model is hierarchical as illustrated in Fig. 3.4 for the SEQ. In each composition, encapsulation of computation is preserved since components do not call one another.

To summarise, the key elements of the X-MAN component model are (1) computation; (2) control; and (3) data. Computation units encapsulate computation; composition connectors encapsulate control; and components encapsulate their own relevant data. Encapsulation of control and computation (and data) allows these elements to be entirely separated. The following sections present the key elements of the X-MAN component model, namely computations and control.
3.5 Key Elements of X-MAN

The following sections describe the key elements of the X-MAN model, namely computations, control and data. For each of the elements, the corresponding relevant architectural constructs will be assigned and discussed. This deliberation is significant during the mapping of the extracted elements to the X-MAN constructs as further discussed in Chapter 6.

3.5.1 Computations

Computations, from the perspective of the X-MAN, are data transformations or services provided by a component. Eventually, these computations will be implemented as methods in programming language. A representation of the component can be illustrated in Fig. 3.5(a). A component here is considered as an independent piece of software that provides computations. It is defined as “a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”[HC01].

In X-MAN, the basic set of rules for computation units are (1) Components perform computations by providing functionalities; and (2) Components do not invoke any computation outside themselves. It has been defined in the component model that semantically, each component is an independent software unit that encapsulates computation units. According to this rule, there should be no direct communication between the components.

3.5.2 Control

In addition to computations, the following key element of the model is control. Control determines the execution flow for a particular composition. Accordingly, X-MAN defines explicit entities that are exogenous connectors and adapters to
handle control. In general, these Exogenous connectors (EC) can be categorised into three: (1) Invocation Connectors (IC), (2) Composition Connectors (CC) and (3) Adaptation Operators (AO).

EC are architectural constructs which separate the control flows from components. As mentioned in the Section 3.5.1, each component encapsulates computation units. In contrast, the logics of the controls are encapsulated in these connectors. The EC roles are to initiate invocations to software components and encapsulate the control flows between components. By designing a component to separate the computational units and the control parts, components can be viewed as loosely coupled, and thus, can be reusable in other parts of the system or even other systems.

IC are unary connectors, which are used to connect to a single component. Since components do not invoke any computation outside themselves, their main purpose of IC is to provide access to the corresponding components’ computation units. Hence, each component must provide an IC to expose its services to other components. During modelling, the IC should be designed to be at the lowest hierarchy-level of the system architecture as depicted in Fig. 3.6.

All of the CC are n-ary connectors that are used to compose a number of
components. These connectors comprise logic of controls needed to compose components i.e. selection, piping or sequencing. In the previous example, in order to compose the AB composite component, the logic of the control for the composition needs to be defined. The semantics of these connectors will be explained in the subsequent sections.

3.5.2.1 Types of Generic Composition Connector

X-MAN provides three generic composition connectors, namely the Sequencer (SEQ), PIPE and Selector (SEL).

A SEQ defines a sequence control execution between components. This type of connector accesses the components sequentially, according to a fixed order defined in the SEQ. For instance, in Fig. 3.7(a) the SEQ provides sequencing ordering from component A, followed by B and finally, C. During the sequencing, the required methods from the corresponding components will be invoked and executed. Imagine that component A comprises method1 and method2, component B has method3 and method4, and component C contains method5. For example, in order to perform a particular task that needs a sequencing execution to perform a set of methods (e.g.: method1, method4 and method5), this is where the SEQ plays its role. The logic inside the SEQ defines the sequential execution of these methods according to the desired order. Eventually, the control flow returns back to the caller.
A PIPE connector specifies a sequence of control between the components by invoking the methods of connecting components, by obtaining the result from the first component and pipes it to the next connecting component respectively. The connector invokes the first component, obtains the result and then passes the result to the subsequent component. As shown in Fig. 3.7(b), when a method in the composite component \( AB \) is invoked, the PIPE connector invokes methods in \( A \), performs the computations, gets the result and passes the result as an input to component \( B \). The same procedure will be repeated until all the respective components are invoked and at the end, the control flow returns back to the caller.

A SEL defines a branching control that is based on a specified condition and then chooses one of the connecting components to be invoked. For example, in Fig. 3.7(c) the SEL checks the pre-defined condition and if the result returns as expected, then the desired component (where the computation unit is encapsulated), will be executed. For example, in the same figure, a specific method, i.e. method1 in the component \( A \), is aimed to be selected if the condition is satisfied. In this case, the SEL encapsulates the condition, and when the composite component \( ABC \) is invoked, the SEL checks the condition and invokes the component \( A \) through its IC if the condition is satisfied. Lastly, the control returns to the caller.
CHAPTER 3. THE X-MAN COMPONENT MODEL

The AO are unary connectors and do not compose components. This type of connector is a unary connector, hence it only connects to a single component [Eli08]. The AO is used to adapt a component in order to make sure that the component can behave according to the desired system requirements. The main purpose of these operators is to allow the execution of the control defined in the adapters to be performed prior to the invocation of the connecting components. For example, when a component is retrieved from the repository to be composed with another component and adaptation is required to the retrieved component, an AO can be used to achieve this purpose. Further details on AO are provided in the respective subsection.

3.5.2.2 Types of Adaptation Operators

The Adaptation Operators (AO) are not composition operators, as they are not used to compose components. The main role of the AO is significant during the deployment phase. In the early phase, which is during design, components are built and stored in a repository. During deployment, these components can be retrieved for further composition. Moreover, during deployment, the components are designed to be application specific and they are in binary code, without any source code. Hence, if any modification of the retrieved components is needed, this is where the AO can play its role. As mentioned earlier, the purpose of these AO is to allow the execution of the control defined in the adapters to be performed prior to the invocation of the connecting components.

The ideal general practice of Component-based Software Development (CBSD) is that components are restricted from being modified directly. Nonetheless, this is not an ideal achievement. In some CBSD approaches, direct manipulation of the components is allowed, and some refer to it as a glue or wrapper code [MC02, gSN99]. However, these concepts allow a component developer to add
codes or modify the component directly which is, in principle, against the ideal belief of what the CBSD community is trying to achieve.

Accordingly, the AO is defined as a means of allowing additional constraints to be added prior to the invocation of a component. The AO can be organised into two main categories, namely conditional and iterative.

For the conditional category, the component model specifies a guard adapter that is used to evaluate Boolean expressions. If the expression returns the expected result, then the connecting component will be invoked. Otherwise, the control returns to the caller. As an example in Fig. 3.8(a), for the sequencing composition, if the specified condition in the guard obtains the expected result, the desired method attached to a component i.e. component A, will be invoked. Subsequently, the control flow returns to the sequencer and invokes the methods in the other connecting component, B.

Meanwhile, the latter category provides two kinds of iterative loop: (1) conditioned-controlled and (2) counter-controlled loop. A conditioned-controlled loop specifies the repetition of the components’ invocations which are defined in the generic EC. If a condition expression specified in this loop returns the expected result, then a generic EC will be performed repetitively until the condition changes. This loop must only be defined on top of the EC level. This rule states that if iteration is
allowed for this particular component, this will produce a more restricted component. In other words, it is more difficult to reuse this adapted component with such restriction. Fig. 3.8(b) illustrates that if the specified condition in the loop adapter returns the expected result, then the SEQ will be executed repetitively until the result of the evaluation changes.

On the other hand, a counter-controlled loop defines a specific number of repetitions of the components' invocations. If a condition expression defined in the loop returns the expected result, then a generic EC will be performed repetitively, according to the counter's value. In order to allow for the possibility of reuse, this adapter must also be defined on top of the EC level. Although this type of loop can also be defined using the conditioned-controlled loop, this counter-controlled loop is defined for simplicity purposes. By having a specialised type of iteration, designing a straightforward system architecture will be much more efficient. The same Fig. 3.8(b) can, in general, illustrate that if the condition specified in the loop adapter returns the expected result, then the SEQ will be executed repetitively according to the value of the counter.

3.5.3 Composition

A composition describes the mechanisms for building a larger composite from atomic components. (See Fig. 3.5(b)). A composition scheme should preserve the component encapsulation. For example, given two components, A and B, if both of these components are composed together, a composition will be created, i.e. AB component. As the proposed approach is defined for the deployment phase, that is the construction of the system is for a specific application only (based on a set of requirements), designing for reuse is irrelevant. This composition of AB should be able to preserve the encapsulation of A and B components. If component
A comprises two computations and component B comprises three computations, the resulting composition of AB should preserve all the five computations of the composition.

3.6 Intrinsic Properties of the Component Model

This model offers three properties, namely component encapsulation, compositionality and reusability. In order to discuss the strengths of this model, some background to the existing component models in the literature will be briefly introduced. In the current component models [LW07, LW06], there are two categories: (i) models which treat components as objects as in object-oriented programming (OOP), and (ii) models which treat components as architectural constructs as defined in software architecture (SA). Each of these categories uses different component composition techniques.

Models in the first category (that are included in the survey, i.e. EJB, COM, CCM, .NET, Web Services and KobrA) use direct message passing to connect the components. The direct message passing mechanism allows a component (or actually an object) to directly invoke any method in other components. In contrast to this, models in the second category (that are included in the survey, i.e. ADLs, Koala, SOFA, PECOS, Fractal and UML2.0) adopt an indirect message passing mechanism, where message passing is done using a mediator element and the mediator will in turn invoke the required methods, whilst for the indirect message passing, the connectors only pass controls from one component to another. As defined in the X-MAN component model, ECs not only compose components, but also encapsulate the logic of the control flows.
3.6.1 Component Encapsulation

By means of separating the computations and controls, the component encapsulation can be preserved. The model defines the component as containing only computations and data, while the controls are handled by the ECs, which are outside of the component. The ECs originate the control and manipulate the control flows among the connecting components. With this view, it is worth noting that the X-MAN model provides a clear separation of computation and control that makes components truly independent. This is not the case in other component models. Models in the first category [LW07], which adopt the direct message passing mechanism, permit components to invoke methods in other components in order to perform a task. Thus, it is unquestionably evident that the components do not encapsulate either computation or control in this regard.

3.6.2 Compositionality

A composition mechanism defines the way in which two or more components can be assembled as a composite component. Because the component has a self-similarity feature, each component is able to be composed to form a larger composite. The X-MAN component model defines composition to be handled in both design and deployment phases. By having these options, a component developer has flexibility in determining whether a component has any potential for reuse or the component is built for a specific application, in which reuse is not the main concern. During design, the main concern is to maximise reuse. If the component developer decides to reuse the component, he or she should store the component in a repository. Meanwhile, during deployment, the required components will be retrieved from the repository, the components will be assembled with other components and compiled them into binaries.
3.6. INTRINSIC PROPERTIES OF THE COMPONENT MODEL

3.6.3 Component Reusability

The significant strength of this component model is the support for reusability in the design phase. The reusability can be achieved with the existence of a repository, where the created components are stored for future reuse purposes. During design or deployment, these components in the repository can be retrieved for further composition. The current component models (e.g. JavaBeans, ADL, EJB, Koala, etc.) do not support composite components in the design phase. This means that composite components cannot be created and reused in the design phase. In the first category of the existing component models (e.g. JavaBeans, ADL, EJB, Koala, etc.) [LW07], the implementations of message passing are done by direct message passing.

The drawback of this mechanism is that it makes the components highly coupled to other components. These components are difficult to reuse, considering the dependencies that are linked among the components. If a developer is considering re-using the components, all the dependencies of the components must also be considered. In addition, there is no explicit code for connectors that can be reused in the semantics of the existing component models. In these models, components are coupled, which makes the components particularly arduous to be reused without considering all the relevant dependencies. While in the indirect message passing technique, composite components are supported during design where each architectural unit is composed by connectors. In this case, it is still difficult to achieve component reuse because the connectors are just passing control from one component to others.
3.7 A Simple Bank System Example

The Bank System example comprises a number of banks associated with a bank consortium (BC). Each bank provides banking services to its clients i.e. deposit, withdraw and check balance. Fig. 3.9 illustrates the bank example using the X-MAN component model.

Since the banks provide services that are the computation units, each bank can be modelled as an atomic component (i.e. BANKA and BANKB). Each component encapsulates CUs and ICs. In Fig. 3.9, the small black circle represents the IC.

For brevity purposes, the internal CUs and the IC details are not shown in the figure. The invocation of the system starts from the top level connector, which is the PIPE. The control flow proceeds to invoke the required computation (operation) in the BC component, performs the computation, and returns the output to the PIPE. The computation to be invoked by the EC has already been defined prior to the execution. The output will then be used as an input to the other branch of the connector. This time, another connector of type SEL is required to decide the correct bank component to be invoked. If the condition defined in the SEL is satisfied, the control flow will proceed to the corresponding component. In this case, imagine that the condition is satisfied, then the flow proceeds to invoke the BANKA component. After the execution of the respective CUs in BANKA, the flow returns to the SEL.

The SEL contains the required constraint to decide an operation to be selected, according to the bank account information. In this particular example, the SEL
connector, which is an exogenous CC, composes both the BANKA and BANKB components. As a result, the BC-C component is created using the CC. And finally, the flow returns to the top level connector. If this is the whole final system, the point of execution is from the top level connector. Meanwhile, if the resulting composition is to be further composed, the top level connector is the point of invocation from the external.

Although there are a few other examples that are used to illustrate the component model (e.g. Automatic Train Protection System [LLW06], and Drink Vending Machine System [LIV07]), the simple Bank System example has sufficiently covered all the generic elements defined in the component model.

### 3.8 Summary

This chapter has provided sufficient background to the X-MAN component model. The fundamental constructs of the model, namely computations, control (and data) have been presented, so as to provide a further in-depth understanding of the subsequent chapters. The main feature supported by the model towards realising the aim of the proposed approach is the incremental composition mechanism. The following chapter explores, presents and justifies the rules for incremental composition to be applied in this study.
Chapter 4

Incremental Composition

The intrinsic feature of the X-MAN component model that enables construction of component-based systems piece-by-piece is the *incremental composition* mechanism. The proposed approach to constructing component-based is incremental, which deals with a single requirement and derives a partial architecture for each requirement in each increment. This is in contrast to work that takes into account all the requirements at once (e.g. \([vL03]\)), including incremental architecture design, which incrementally adds behaviour or properties to an architectural skeleton (e.g. \([BDLM05]\)) as discussed in Chapter 2. This work is also different from the work that incrementally develops requirements **hand-in-hand** with architectures (e.g. \([RHJN04]\)). This chapter presents, discusses and justifies the incremental composition rules to be applied in the approach.

4.1 Introduction

To use a component-based approach for building systems directly from requirements, a suitable component model should be adopted, in particular one that
supports *incremental composition*. In a component model, a composition corresponds to an architecture, and by incremental composition which defines composition that (i) allows the addition of more components, as well as the addition of further compositions, to an existing architecture; and (ii) preserves the behaviour (and hence properties) of the existing architecture within the incremented architecture.

A simple example of incremental composition is provided by the C2 component model [TMA+95]. This is illustrated in Fig. 4.1(a). In C2, components are composed using a bus connector. These components are attached to a bus which communicates indirectly with each other via the bus. The bus receives messages from the connected components and broadcasts them to all the components. These components identify messages that they can handle and respond to accordingly. Components in C2 are active and connectors are passive. Compared with exogenous composition connectors in X-MAN, C2 connector does not have any control logic other than broadcasting and control starts from an arbitrary component. Since components do not communicate directly with each other, any number of components can be attached to the bus. Accordingly, more buses can be added to create additional layers in the architecture, as shown in Fig. 4.1(b).

![Diagram](a) Before (b) After

**Fig. 4.1:** Composition in C2.

By preserving the behaviours of the existing architecture, the incremental
composition supports an incremental approach to mapping requirements directly to systems. Requirements can be successively mapped one at a time into a partial architecture by adding further components and/or compositions. Initially, this proposed approach starts with an empty architecture, and increments the architecture with a partial architecture such that it satisfies one requirement.

Following that, for each of the other requirements, the current partial architecture is successively incremented (by adding more components and compositions) such that, each time, the new architecture satisfies the new requirement, together with all the previous ones, by virtue of behaviour preservation. The complete architecture is the final architecture when all requirements\footnote{We focus only on functional requirements.} have been mapped in this way.

Incremental composition allows the addition of components and composition at any permissible point. Nevertheless, in order to preserve all the requirements that the current partial architecture has already met, it is crucial that the incremental composition is done in a manner that is incremental with respect to the incremented behaviours that have already been satisfied.

The semantics of the incremental composition with respect to requirements mapping can be expressed as the relations in Fig. 4.2, where Rs are requirements, $S$s are partial architectures, and $\subseteq$ denotes the ‘subset of’ or ‘is contained by’ relation. Here, the ‘$\subseteq$’ notation is loosely defined: $\{R_1, \ldots, R_n\} \subseteq S$ means partial architecture $S$ satisfies the set of requirements $R_1, \ldots, R_n$; and $S_1 \subseteq S_2$ means partial architecture $S_2$ contains partial architecture $S_1$.

\[ \{R_1\} \subseteq S_1 \]
\[ \{R_1, R_2\} \subseteq S_2 \]
\[ \vdots \]
\[ \{R_1, R_2, \ldots, R_n\} \subseteq S_n \]

Fig. 4.2: Incremental composition.
4.2 A Component Model with Incremental Composition

The X-MAN component model supports incremental composition. In addition to the fundamental concepts of the X-MAN component model, as presented in Chapter 3, this section further discusses how the model supports incremental composition. In the X-MAN component model [LVW05, LOW06, LLW06], computation and control are encapsulated separately. This separation and encapsulation enables the requirements to be mapped to partial architectures, according to the X-MAN model, by identifying computation and control specified in requirements and mapping them to corresponding elements in the model. Because of this feature, the mapping process can be undertaken without having to analyse and check for all the components’ dependencies as in other component models such as ADL-like or object-oriented models.

In X-MAN, components encapsulate computations and data, whilst composition connectors encapsulate control. These components have no external dependencies, and can therefore be depicted in Fig. 4.3(a) and (c), with just a lollipop (provided service), and no socket (required service).

Fig. 4.3(a) shows an atomic component that consists of a computation unit (CU) and an invocation connector (IC). A CU contains a set of methods which has no direct access to methods in the computation units of other components; it therefore encapsulates computation. An IC passes control (and input parameters) received from outside the component to the CU to invoke a chosen method, and
after the execution of the method passes control (and results) back to where it came from, outside the component. It therefore encapsulates control.

Fig. 4.3(b) shows a composition connector, which encapsulates control structures, namely sequencing, branching or looping execution. The components as defined in the X-MAN component model therefore encapsulate control (and computation) at every level of composition\(^2\). Clearly, composition in the X-MAN model is hierarchical and it preserves encapsulation at every level.

Fig. 4.3(c) shows a composition of two components CardReader (CR) and PINReader (PR), composed by a SEQ composition connector. The control starts when the customer enters his card and follows with a request to enter his PIN. The connector SEQ passes control from CR, which reads the customer’s card; then it passes control to PR, which reads the customer’s PIN. Control then passes back to the top SEQ and the control is ready to be passed to the following execution.

As a brief summary from Chapter 3, other composition connectors in the X-MAN model include PIPE for sequencing (it is the same as SEQ except a PIPE passes the results from one component as input to the next), and SEL for branching (it selects one of the connecting components). The X-MAN model also defines unary connectors which act as adaptors for composition connectors: loop for looping,\(^3\) and guard for passing or inhibiting control flow to a composition connector.

In order to support incremental composition, (i) composition connectors are allowed to be open in arity\(^4\), thus allowing any number of components to be added to an existing composition connector; (ii) composition are allowed to be open, i.e. to have open or incomplete interfaces, as shown in Fig. 4.4.

A composition connector is open by default; the ‘...’ adjacent to an open composition connector in Fig. 4.4 denotes available composition points, i.e. points

\(^2\)They also encapsulate data at every level of computation [LT06].
\(^3\)All loops must be finite, except for a loop at the top level of a system.
\(^4\)The number of arguments i.e. the components, that the connector takes.
where more components or compositions can be added. A closed composition connector (e.g. the ones in Fig. 4.3(a) and (c) that are denoted by solid lolly-pops interface) in contrast does not have any available composition points. An open composition connector can be closed (i.e. can become a closed connector) by simply removing its available composition points; this is a change in property, and can be introduced manually. In contrast, a closed connector cannot become open. An open composition connector creates an open composition with an open interface, which is denoted by a hollow lollipop circle ((a)(i), (b)(i), (c)(i) and (c)(ii) in Fig. 4.4), whereas a closed composition connector yields a closed composition with a closed interface, which is denoted by a solid lollipop circle ((a)(ii), (b)(ii) and the composition of C and D in (c)(ii) in Fig. 4.4). In addition, all the open composition points denoted by ‘...’ are removed from the compositions.

An open composite can be closed by closing its composition connector, but only if all its sub-components are closed. In Fig. 4.4, (a)(ii) and (b)(ii) are closed when the composition connectors in (a)(i) and (b)(i), respectively, are closed (all their sub-components are already closed). Thus closing a composition is done hierarchically, from the bottom up.
In Fig. 4.4(c)(i), the top open interface can only be closed after the open interface for the composite containing \( C \) and \( D \) has been closed (as in (c)(ii)). Due to encapsulation in components, and hierarchical composition that preserves encapsulation, the composition in Fig. 4.4 satisfies the relations in Fig. 4.2 (considering open compositions as partial architectures), i.e. it is indeed incremental composition. Encapsulation ensures that newly added components do not alter the behaviour of existing components, and hierarchical composition preserves requirements that have already been satisfied by the current partial architecture.

4.3 Incremental Composition in Existing Component Models

A number of works in the literature, with regards to what is defined as incremental composition have been proposed for handling better flexibility during composition. In this section, these works are considered and discussed. In addition to the C2 model, the discussion in the context of incremental composition includes generic Architecture Description Languages (ADLs), Invasive Composition, Aspect-Oriented Composition and Behaviour Engineering (BE).

4.3.1 Architecture Description Languages

In any generic ADL, the model defines components as architectural units with exposed provided and required ports as depicted in Fig. 4.5. Components can be composed with other components to build a composite component, by connecting the required ports with the compatible provided ports. This means that composition happens when connections between these ports are established.
4.3. INCREMENTAL COMPOSITION IN EXISTING COMPONENT MODELS

Imagine that there are four ADL components, i.e. A, B, C and D, with its corresponding provided and required ports. The composition of these components happens if there is at least one connection between each component. As such, the *arity* of the composition is either one or many, but the possibilities are fixed based on the matching required and provided ports. Hence, for each composition (using connection), all the dependencies via the required ports have to be checked.

4.3.2 Aspect-Oriented Composition

The aspect-orientation approach is also claimed to provide a means of flexible composition [CHS08]. The approach uses *aspects* and *point cuts* to define behaviours that share cross-cutting concerns among the rest of the code. *Join points* are the events during execution at which *aspects* may execute, whereas a *point cut* is a set of join points [WKD04]. These aspects are woven into specific locations addressed by the point cuts as briefly visualised in Fig. 4.6.
An aspect weaver generates the required glue codes in order to compose the aspects into an application system. Nonetheless, in the balancing act of providing flexible composition and achieving reusability, the use of point cuts in aspect-orientation does not guarantee behaviour preservation. Hence, this mechanism is not an ideal property for supporting incremental composition that we seek for.

4.3.3 Invasive Software Composition

Invasive software composition [Ass03] adapts and extends components using hooks by transformation. Hooks are variation points of a component; fragments or positions which are subject to change. Invasive software composition defines two types of hook: (1) implicit hooks and (2) declared hooks that are represented as small boxes on the interface of a component, as shown in Fig. 4.7.

![Diagram of hooks in invasive software composition](image)

(a) Hooks in invasive software composition
(b) Composition technique

Fig. 4.7: Invasive Software Composition

In invasive software composition, a composition occurs when a component or fragment is *weaved* into another component with declared hooks and thus the composition operator generates the required code. In this case, the composition operator has a fixed *arity* which means a specific number of component(s) or fragment(s) can be composed and the composition operator (i.e. the operator in Fig. 4.7(b)) will generate the code as a result of the composition. The composer
can be used for coordination, inheritance and distribution of aspects over core application. Invasive software composition does not preserve the existing behaviour in each component, but generates the codes to glue the composition instead. When these glue codes are added, the components are modified accordingly. For this reason, by referring to the definition of the incremental composition, invasive composition does not adhere to the defined concept.

### 4.3.4 Behaviour Engineering

One of the sources of work that is considered to be relevant is the behaviour Engineering (BE) approach that defines tree-like structure to represent behaviours of entities, which is called Behaviour Tree (BT) [Dro05, Dro03b, MD09]. BT provides a mechanism to support flexible composition with some constraints during one of the stages which the authors refer to as the integration process. During this process, each connected node, i.e. the requirements behaviour tree (RBT) will be incrementally integrated with a single design behaviour tree (DBT) by using some guidelines. The results of the integration process are either as depicted in Fig. 4.8(c) or Fig. 4.9(d).

![Fig. 4.8: The Result of a Sequencing Composition.](image-url)
In BE, the software composition units according to the definition of the incremental composition are the RBTs and the executional semantics which are either a sequential, branching or looping operator. Fig. 4.8(c) depicts the result of a sequential composition, whilst Fig. 4.9(d) shows the result of a branching composition. Composition occurs when two RBTs are integrated into a single DBT. In their work, only two RBTs are integrated at a time except for cases where it is not clear where to integrate those RBTs [Dro05, Dro03b, MD09].

![Diagram](image)

**Analysis** In current component models as discussed in [LW07], only Koala [OvdLKM02] supports a form of incremental composition by preserving properties defined in the interfaces of the sub-components. Koala provides a specialised interface, called the “diversity interface”. This interface, by definition, enables the properties of sub-components to be propagated to parent-components until the system interface has been reached.

In other component models, the entire system is designed and/or constructed in one step, either at the design phase (as in Enterprise JavaBeans [DK06] and UML2.0 [OMG]) or in the deployment phase (as in JavaBeans [Ham97]).

Using JavaBeans, in the design phase, new (atomic⁵) components can be deposited in a repository, but cannot be retrieved from it. Composition is impossible

---

⁵Components that are not constructed from other components.
in the design phase: that is, no software architectures can be specified. Accordingly, an architecture is created and instantiated from atomic components, which are retrieved from the repository, only in the deployment phase. In this regards, JavaBeans does not support incremental composition. A system architecture that supports given requirements is specified and instantiated from components retrieved from BeanBox⁶ (JavaBeans repository). Properties supported by an architecture are all decided in one step and can never be enriched in a subsequent composition operation.

In PECOS and UML2.0, in the design phase, there is no repository. Hence, architectures are devised and specified from scratch, meaning that composition is possible. In the deployment phase, no new composition is permitted. The composition of the component instances (in the runtime phase) is the same as that of the components in the design phase. The Design without Repository category as defined in [LW07] does not allow for the building of a system architecture from partial architectures; the entire architecture is specified in a single step. Thus, this category does not support incremental composition.

4.4 Design Decisions for Incremental Composition

This section elaborates and discusses the design decisions to apply incremental composition cases. Based on examples, the incremental composition cases are motivated and justified. Following this set of design decisions, the designer could justify his or her design alternatives and selection. The final outcome of this approach is a system architecture. In the end, only a single architecture will be produced. A system analyst’s or a system designer’s task is to somehow link all the requirements to achieve the final architecture. When the architecture is

⁶http://wiki.netbeans.org/NetBeansJavaBeansTutorial
incrementally built, decisions must be made as to which composition point can be used to compose the current architecture with the partial architecture. The first derived partial architecture will automatically be considered as the initial system architecture. This section explains the basis on which a composition point is determined.

To decide on the incremental composition, a system analyst or a system designer needs to consider two factors: (1) type of composition or adaptation and (2) type of connector or adapter. First, he/she needs to analyse whether the composition or adaptation is selection-based, ordering-based, condition-based or repetition-based. Such an analysis is important in deciding on the application of the composition points in order to sustain the aims of incremental composition.

The second consideration is the type of connector or adapter to be applied at the chosen composition point. A composition point is a valid location (syntax and semantics) where the new composition can occur. If the composition point has the same type of required connector (i.e. homogeneous), composition can be either at the first, middle (only for SEL) or at the end. In contrast, if the composition point has different types of connector (i.e. heterogeneous), he/she has to compose using a new composition connector, and if the behaviour cannot be represented, the alternative is using refactoring technique. This technique will be presented in Chapter 7.

![Composition points diagram](image)

Fig. 4.10: Composition points.

A valid composition point is the position where a new composition is allowed
to happen in order to devise a meaningful composition. As such, even if syntactically, an architecture comprises more than one available composition points (denoted by adjacent of ‘...’), a valid composition point corresponds to the correct representation of the intended behaviours from the requirement.

In Fig. 4.10, although there are many available composition points in each hierarchy level, only the valid composition points are considered. In the following subsections, the fundamentals of design decisions that are applicable while applying incremental composition are addressed. These design decisions are categorised into (1) ordering-based composition, (2) selection-based composition, (3) condition-based adaptation and (4) repetition-based adaptation.

### 4.4.1 Ordering-based Composition

This section provides the design decisions on the application of the incremental composition for ordering-based composition. The design decisions will be supported and justified by examples.

**Composing a new component or composition to a composition point.**

When a logic corresponds or relates to an existing ordering execution that has been modelled, e.g. in the example AB (Fig. 4.11(a)), it is logically correct to be piped or sequenced with C, so that C can be composed to the end of the composition point. Fig. 4.11(b) illustrates the composition. Alternatively, if the logic denotes composition that happens at the front of the composition point, composition can be designed as in Fig. 4.11(c). By restricting composition (either at the front or at the end) for ordering-based composition, we adhere to the incremental composition by preserving the incremented behaviours. In the example, the ordering execution of AB is preserved by restricting new composition to be
only at the front or at the end of the composition point. There is no anticipated issue when dealing with such a composition using a SEQ.

![Diagram of composition](image)

Fig. 4.11: Adding a component to an existing composition.

However, a potential risk with such a case is when dealing with PIPE connectors. This is mainly due to the fact that PIPE connectors comprise data dependencies. When composing new components to an existing PIPE connector, design decisions for handling and providing data must be resolved. Otherwise, some of the connected components might not receive the expected or required data, due to the missing passing data or outputting data. A foreseen issue is when a new component is composed at the front of an existing composition point. Such a case may lead to unavailability of data from the first connecting component because originally the component does not anticipate any input data from any component.

For example, component A in Fig. 4.11(a) is originally the first component, which does not require any input from other components. When a new component, C as in Fig. 4.11(c) is composed to front of the composition, an output from C will be piped to A, whereas in the original execution of A, no input is anticipated. The possible solution to address this issue is provided in the discussion part of this chapter. See Section 4.5.3.

**Example 1**

Consider an example from the ATM system. Imagine that the following composition comprises Card Reader (CR) and Pin Reader (PR) components with a
PIPE. The result of this composition is piped to another composition of Authentication (AUT) and Withdrawal (WD) components (see Fig. 4.12(a)).

**Analysis.** The question here is why do we use a new PIPE instead of directly extending these new components to the existing PIPE? This decision is based on whether the need to iterate a specific component is being considered or not. If the answer is yes, breaking components into a smaller logical structure will be useful. Without this consideration, an iteration for that specific composition unit cannot be modelled except for the whole composition.

![Fig. 4.12: ATM example-1.](image)

Consider if a new bank transaction, Deposit (DP) is to be composed with the existing composition (see Fig. 4.12(a)). Based on the semantics of the X-MAN component model, two composition points where DP can be composed are either the second level PIPE or the right side PIPE at the first level, but definitely not the left side PIPE at the first level, as in order to perform the deposit transaction, the user needs to be authenticated beforehand. In this context, ordering is deliberated as a criterion during composition.

**Example 2**

Consider an example from Word Count. In the Word Count example, once the file is validated, the words in the file will be formatted according to the type of input file. Otherwise an error shall be displayed.

**Analysis.** Fig. 4.13(a) illustrates the behaviours of file validation and error
CHAPTER 4. INCREMENTAL COMPOSITION

Fig. 4.13: Word Count-2

display, whilst Fig. 4.13(b) illustrates the behaviours of file validation and file formatting. Consider if both of these compositions are derived in a different incremental step. If these two compositions are to be composed together, while maintaining the same behaviours, the VF component does not need to be repeated. The alternative design is to maintain the composition of VF-DE and later on compose FF component to the end of the composition together with its guards. The derived architecture after IC is shown in Fig. 4.13(c).

Example 3

Consider another example from ATM in which a Transfer Fund (TF) component is composed to the existing composition.

Analysis. Now, suppose a new requirement stating that the account from which the funds are transferred must not be overdrawn. This particular computation must be checked prior to the transfer fund transaction. Hence, ordering is a consideration here.

Fig. 4.14: ATM Example-3.
Example 4

Let us say the ATM requirement states that for each transaction the ATM system will print the corresponding receipts containing all the relevant transaction information.

Analysis. Fig. 4.16 contains a two-level hierarchy of connectors. The decision to allocate the new composition unit is based on (1) type of composition or adaptation and (2) type of connector. The type of composition is determined by the execution order. For instance, in the composition illustrated in Fig. 4.15, it is wrong to place the Reject Card (RC) component next to the PIN Reader (PR) component because the flow of logic clashes (authentication must be performed prior to the rejection).

As the same type of connector, i.e. PIPE, is used, the new composition unit can be composed either at the end of the first-level hierarchy or the top level hierarchy.

If a different type of connector is being applied, any new component certainly cannot simply be composed at any of the available composition points.

4.4.2 Selection-based Composition

This section provides the design decisions for selection-based composition. Because ordering is not significant for this type of composition, a new component
or composition may be added at any valid composition point.

**Composing a new component or composition to a composition point.**

For selection execution, although the location where the component is added does not make any difference, for consistency, a new component or composition could be added at the end of the valid composition point. In other words, a new component or a composition can be composed either at the front, in the middle or at the end of an existing composition. This condition is applicable to any generic Selector (SEL) cases. Fig. 4.17(a) depicts the original composition of A and B components with a SEL connector. In order to compose a new component to the existing A and B composition, a new composition (i.e. component C) is allowed to be composed either at the front (see Fig. 4.17(b)), in the middle (see Fig. 4.17(c)) or at the end (see Fig. 4.17(d)) of an existing composition.

![Diagram showing compositional relationships of components A, B, and C](image)

Fig. 4.17: Composing to a new component to a composition point.

The only condition for such a case is to re-consider the selection constraints which are encapsulated by the SEL connector. Without this amendment, the
newly added component will never be reached.

**Example 5**

Consider an example from Automated Library Circulation System (ALCS)\(^7\)

**Analysis.** If a selection of ALCS tasks i.e. adding books (AB) and removing books (RB) is to be composed. As ordering is not a significant consideration for composition of the AB-RB, there are two possibilities: either to compose them as (1) **AB-RB** (Fig. 4.18(a)) or as (2) **RB-AB** (Fig. 4.18(b)).

Both of these options are able to capture and represent the intended behaviour from the requirement.

Consider if the succeeding requirement states that a user can select to check out books or remove books. Fig. 4.19(a) illustrates this feature. The following discussion explains how this architecture can be further composed using the incremental composition.

The preceding architecture (Fig. 4.19(a)) contains the *selection-based composition* of checkout (CO) and return book (RTB) transactions. The subsequent requirement adds another composition of add(AB) and remove book(RB) transactions. When dealing with this type of composition, composing **AB-RB** (by extending the open composition of the top level **SEL**) can either be at the front, in the middle or at the end of the existing composition point. Nonetheless, in

\(^7\)Taken from [Sch06, p. 246].
this case, for consistency, the composition is added at the end of the composition point. This is shown in Fig. 4.19(b). It is important to note that the selection criteria for the \textsc{SEL} must be amended by adding new constraints for the newly added component or composition. The derived architecture can be further simplified; however, a discussion on architecture refactoring will be provided in Chapter 7.

### 4.4.3 Condition-based Adaptation

So far, the incremental composition cases that have been discussed are compositions that apply to ordering-based and selection-based executions. The following subsections now discuss the roles of incremental composition when it comes to adaptation. In Chapter 3, the two main types of adapters have been elaborated, namely the \textit{guard} and \textit{loop} adapters. Both of these adapters are \textit{unary} connectors, which means they are used to adapt a single component, and not used for compositional purposes.
4.4.3.1 Adding a guard adapter to a component.

Guards\(^8\) are used to check certain conditions before performing any execution provided by the connecting component. In Fig. 4.20(b), guard is denoted by a triangle. If the conditions are satisfied, only then will the intended computation be invoked, otherwise the control returns.

A foreseen issue with adding a guard to a component is that once the guard is added, the condition to the subsequent branches should be propagated (brought forward), which makes it difficult to handle. That is why the aims of architecture refactoring (see Chapter 7), are to reduce the number of guards and at the same time simplify the design. Nonetheless, this case is only applicable to handling PIPE connectors with guards because a guard contains constraint data. This constraint data must be provided during execution. Without the data, the condition encapsulated in the guard cannot be evaluated. As a result, the intended computations may not be invoked.

Example 6

Consider another example from the ATM System. For the sake of argument, consider if the Authentication (AUT) and Withdraw (WD) components are composed. The result of the authentication computation will be passed to the connecting component i.e. the WD component to allow an amount of money to be withdrawn.

\(^8\)Guards are *unary* connectors.
Analysis.

However, if there is no validation before the execution of withdraw computation, the invocation can still happen even if the authentication fails. In order to restrict the control to proceed with such invocation, a guard is needed to adapt the WD component (see Fig. 4.21(b)). By adding a guard here, the result that is piped from the preceding component, i.e. AUT, will be checked prior to the invocation of the connecting component’s computation.

4.4.3.2 Adding a Guard to a Component (in between two components)

By adding a new guard in between an existing composition, data to be used to check for the condition must be available during execution. Normally, the preceding connected component shall provide the data to be evaluated by the guard. Thus, the only possible connector for CC here is a PIPE as only PIPE is able to pass data to be evaluated as a constraint in this case. If a SEQ is used, then adding a guard to the composition will not be useful.
4.4.4 Repetition-based Adaptation

For a loop adapter, it is crucial to ensure that the loop will eventually terminate, in order to be appropriately applied to a composition. Imagine if the condition never terminates; the whole composition will be problematic. As such, cautious decisions are required when dealing with loop adapters. Based on Fig. 4.23(a), a loop adapter is adapted to the existing AB composition, and the result is shown in Fig. 4.23(b). Note that the loop adapter is added on top of the AB composition.

Example 7

Consider an example from the Trading System (TS). In the card payment transaction, the Cashier receives the credit card from the Customer and pulls it through the Card Reader. The Customer enters his PIN using the keyboard of the card reader, and waits for validation.

Analysis. Here, the Card Reader (CR) component is composed with the AUT component, as depicted in Fig. 4.24(a). Let us say the succeeding requirement states that the steps are repeated until successful validation or the Cashier presses the button for cash payment. The next increment is to add a loop adapter to the composition, as illustrated in Fig. 4.24(b). The condition in the loop must satisfy the stated constraints from the requirement.

\[^9\text{Taken from [RRM08].}\]
CHAPTER 4. INCREMENTAL COMPOSITION

Example 8

Consider an example from the ATM system. Suppose that the following composition comprises Card Reader (CR) and Pin Reader (PR) components with a PIPE. The result of this composition is piped to another composition of Authentication (AUT) and Withdrawal (WD) components (see Fig. 4.25(a)).

Analysis. The decision to choose a correct composition point for this example is based on whether the need to iterate a specific component is being considered or not. If the answer is yes, breaking components into a smaller logical structure will be useful. Without this consideration, an iteration for that specific composition unit cannot be modelled, unless for the whole composition.

Fig. 4.25: Repetition-based composition.

4.5 Issues and Discussion

This section presents the potential issues during incremental composition.
4.5.1 Combining redundant behaviours

When some parts of the architecture have already been represented in the current architecture, any later parts of architecture can reuse any useful parts of this architecture. By ignoring this rule, it might be difficult or there may even be no other way to represent the desired behaviours.

For example, the existing design comprises an AUT component that provides authentication computation. In the later increment stage, another hint leads to the same authentication process, which has the same behaviour as the one that had been represented in the former increment. By combining the redundant behaviours, the former AUT component can be reused for other branches of execution while attempting to maintain the behaviour that is required.

4.5.2 Amendment of constraints involved during IC

For selection-based composition, amendment to the respective constraints needs to be handled. An implication of this is the possibility that the newly added component will never be accessible. Each time a new component or a composition is composed using a SEL accordingly. The constraints and their selection criteria must be supplemented.

4.5.3 Consideration of piped data

As mentioned in Section 4.4.1, when dealing with PIPE connectors that have data dependencies, design decisions for handling and providing data must be resolved. As a consequence of the missing passing data or outputting data, some of the connected components might not receive the expected or required data that is crucial for its computations. A foreseen issue is when a new component is composed at the front of an existing composition point. Such a case may lead
to unavailability of data from the first connecting component, because originally, the component does not anticipate any input data from any component.

These issues must be handled properly in order to overcome design problems. In some situations, empty (or pre-defined) values are passed to satisfy this condition. In addition, PIPE connectors can be devised so that the outputting data will be piped to the rest of the connecting components and only the required data will be conceived by the connecting components throughout the execution.

4.6 Summary

This chapter has elaborated and discussed the incremental composition and how this mechanism may be applied during composition of the extracted component-based elements into an architecture. The incremental composition preserves the incremented behaviours and at the same time propagates the services to be further composed. The factors that influence the design decision on the incremental composition are (1) based on the execution order that is either selection-based composition, ordering-based composition, repetition-based adaptation or condition-based adaptation; and (2) based on the type of connector or adapter. For each of the incremental composition cases, design guidance and examples are provided.
Chapter 5

Extracting Elements of Component-based Systems from Natural Language Requirements

In dealing with raw requirements written in natural language, an analysis of the literature has been undertaken to investigate how analysts and designers decipher and extract information from requirements. The result of the analysis in Chapter 2 has addressed two means of handling requirements, either by analysing the entire set of requirements or dealing with each requirement. In the mission to incrementally build an architecture, accordingly it is also necessary to deal with requirements incrementally, which means analysing them one at a time. This work provides the means to deal with requirements by extracting component-based elements directly from NLR using a textual analysis technique. This section starts by presenting the related work on textual analysis techniques in the literature. Following this, a set of rules for the element extraction process grounded by the fundamental concepts of the X-MAN component model is justified and proposed.
CHAPTER 5. EXTRACTING ELEMENTS OF CB SYSTEMS

5.1 Introduction

A key step in this process of constructing component-based systems incrementally is the extraction of keywords (from NLR) that correspond to elements of the chosen component model. Once such elements have been identified, creating a component-based system only requires them to be pieced together according to the selected component model semantics.

5.2 Related Work on Information Extraction

Extracting keywords from natural language requirements is not a new idea. Indeed, it has been practised for a long time. However, none of the existing techniques has been designed or used for extracting keywords that correspond to elements of component-based systems, i.e. systems defined using component models. Existing techniques have been used to extract keywords that map to abstract concepts, intermediate requirements models, object-oriented analysis models and even to skeleton programming language constructs.

<table>
<thead>
<tr>
<th>Element Extraction Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>map to abstraction concept</td>
</tr>
<tr>
<td>AbStFinder[G97]</td>
</tr>
</tbody>
</table>

* object oriented models, structured models, database models
** based on object oriented

Fig. 5.1: Related work on Information Extraction
5.3. IDENTIFYING COMPONENT-BASED ELEMENTS FROM NLR

[SHE89, Boo86, JML00, Mic96, Bro01, DBB99, HG03, OBO01] extract elements based on intermediary requirements models such as object-oriented models (e.g. Unified Modelling Language (UML) class diagram, object diagram), structured models[AG97] (e.g. Data Flow Diagram (DFD)), database models (e.g. Entity Relationship Diagram (ERD), Extended Entity Relationship (EER)[Che80, HL07, TB94, DM06, AG97]). As such, the elements to be extracted are identified based on the concepts defined in the respective models.

This category classifies forward engineering approaches that adopt intermediary requirements models, because these models are used to be further refined or detailed into more design models and eventually devise an architecture diagram for a particular system. Apart from this category, work such as Larman [Lar04] and AbstFinder [GB97] extract elements to identify terms as candidates for object-oriented concepts. This process aims to assist in the requirements elicitation process, rather than building a working system. In contrast to the category of approaches that derive intermediary requirements models, [Abb83] extracts elements (e.g. data types, operators and control structures) that are used to specify pseudo codes. Based on our knowledge, there is no work that (1) directly maps NLR into component-based constructs and (2) is motivated by a specific component model as the basis of extraction elements.

5.3 Identifying Component-based Elements From NLR

Rules represent definite assumptions which are pieces of syntactic information extracted from natural language statements, whereas heuristics represent indefinite assumptions that allow us to reason during the mapping of the extracted elements.
into the corresponding desired modelling elements [TB94]. Heuristics can also be used as a learning device that can help to gain insights into a complex problem and further, may be modelled and solved more pragmatically [TB94]. Research on the use of heuristics to aid the identification of elements extraction process from natural language has been investigated [Abb83, SHE89, Boo86, JML00, Mic96, Bro01, DBB99, HG03, OBO01, TB94, AG97, Che80, HL07, TB94, DM06, GB97, Lar04, Boo86]. In general, the elements extraction process and its classification of elements to be extracted depend on the underlying applied theoretical concepts. In the literature, this classification is mostly based on object-oriented software development. For example, Booch [Boo86] uses the heuristic that nouns correspond to objects or classes, whereas verbs correspond to messages or methods in object-oriented software development.

5.3.1 Elements Extraction from Natural Language Requirements

Given that the intention of this research is to map requirements to architectures in the X-MAN component model, elements in requirements that correspond to the key semantic concepts in the model need to be identified, and hence, words (in requirements) that represent these concepts also have to be extracted. Most object-based mapping approaches rely to a large extent on identifying nouns and verbs because the object-oriented computational model supports only operands and operators [Boy99]; object-oriented software development [Boo86] uses the rule that nouns correspond to objects or classes, whereas verbs correspond to messages between objects.
As presented in Chapter 3, the key semantic concepts in the X-MAN component model are *computation* and *control*. Computation means data transformation or function evaluation, in which values or functions are computed and variables may be updated. Control means the flow of execution of pieces of computation. Thus, the result of a piece of control invoking a computation is a piece of behaviour\(^1\).

The following tables summarise what can be extracted from natural language requirements. The main elements that can be identified are verbs. Verbs generally refer to actions, events and processes [Jac82]. These verb categories are adopted and adapted from Saeki's [SHE89] and Rolland's [RP92] rules for identifying verbs and mapping them into the X-MAN component model elements. Fig. 5.2 shows what can be extracted from verbs. In Fig. 5.2, *Computation* and *State* categories are adopted from *Action* and *State* [SHE89, RP92] whilst *Event* category is adopted from *Emergence* [RP92].

<table>
<thead>
<tr>
<th>Category (of verbs)</th>
<th>Denotes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transformation</td>
<td>Computation(data transformation)</td>
<td>withdraw, deposit, cooking</td>
</tr>
<tr>
<td>State</td>
<td>Internal state of components (attribute values of components)</td>
<td>keep, remain</td>
</tr>
<tr>
<td>Event</td>
<td>Events that can trigger computation</td>
<td>press, cancel, push</td>
</tr>
</tbody>
</table>

Fig. 5.2: Elements that can be extracted from verbs.

A *computation* verb, e.g. *withdraw*, denotes a data transformation, which takes data as input, performs some function evaluation and outputs data, in order to achieve a specific objective. In general, data transformations can involve simple data access operations i.e. input/output operations; however, these kind of computation verbs are not used to denote such data transformations. Only database operations as computations, e.g. search, updates are considered. Verbs

\(^1\)For simplicity it is assumed that data follows control.
that are physically performed by humans are also excluded as data transformation.

A *state* verb, e.g. *keep, maintain, cooking* denotes computations that realise states, i.e. change the attribute values of components. However, not all *state* verbs can be mapped to a computation. Normally, states which are of interest are ones that can be associated with computations. In order to achieve this kind of state, computations need to be performed. This means that the system should provide corresponding computations. States such as *idle* which may not be doing anything, do not imply any computations.

An *event* verb, e.g. *press*, denotes an event that can trigger computations. This *Event* verb can indirectly imply computations that should be provided by the system. Because the requirements are unstructured, such hints can be a guidance in determining candidate computations.

<table>
<thead>
<tr>
<th>Category (of nouns)</th>
<th>Denotes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual component</td>
<td>Conceptual hooks for components</td>
<td>power tube, authentication</td>
</tr>
<tr>
<td>Data</td>
<td>Value or set of values</td>
<td>1,c.integer</td>
</tr>
<tr>
<td>State</td>
<td>Attribute name of state</td>
<td>closed.open</td>
</tr>
<tr>
<td>Computation</td>
<td>Computation (data transformation)</td>
<td>registration, transmission, movement</td>
</tr>
</tbody>
</table>

Fig. 5.3: Elements that can be extracted from nouns.

Fig. 5.3 shows what can be extracted from nouns. In Fig. 5.3, *Conceptual component* is adopted from *Class*, *Data* is adopted from *Value*, *State* is adopted from *Attribute* and *Action* is adopted from *Action* respectively [SHE89].

A *conceptual component* noun, e.g. *power tube*, denotes an abstraction of a candidate component that can be identified from nouns such as devices (e.g. power tube, auto-teller machine, etc.). In an explicit scenario, computations can usually be directly identified from verbs. This is the general case. Nonetheless, when such *conceptual component* nouns are extracted, they are already associated with its corresponding computations which may not be explicitly specified. Based
5.3. IDENTIFYING COMPONENT-BASED ELEMENTS FROM NLR

on the knowledge of an analyst, he or she may identify specific computations relevant to the conceptual component.

A *data* noun, e.g. the number ‘1’, denotes a value that may need to be stored and retrieved. These values may represent status, grade, result of computation, etc. Thus, the *data* noun category is identified to store meaningful values to be used by computations.

An *action* noun\(^2\), e.g. *registration*, denotes data transformation provided by a component such as authentication and initialisation processes. In the extraction process, which is supported by the Extractor tool (see Section 5.5), elements of the component-based system are suggested based on verbs, nouns or phrases using a textual analysis technique. By excluding the *computation* noun category, the extraction process might not be comprehensive in covering the extraction of all candidate computations.

<table>
<thead>
<tr>
<th>Category (of phrases)</th>
<th>Denotes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptive expression</td>
<td>May denote components or computations</td>
<td>‘the earlier date’ may denote date or compareDate()</td>
</tr>
<tr>
<td>Predicate</td>
<td>Computations - operations that can be true/false</td>
<td>is enabled, is invalid</td>
</tr>
<tr>
<td>Control structure</td>
<td>Control structure</td>
<td>if, then, else, while, iterate, loop, after</td>
</tr>
</tbody>
</table>

Fig. 5.4: Elements that can be extracted from phrases.

Fig. 5.4 shows what can be extracted from phrases. In Fig. 5.4, the *Descriptive expression*, *Predicate* and *Control structure* categories are adopted from *Descriptive expression*, *Predicate* and *English control structure* respectively [Abb83].

In addition to the verb and noun categories, a *descriptive expression* phrase, e.g. the *earlier date*, may denote computations. Abbott [Abb83] specifies that a descriptive expression describes a possible object whose identity (and possibly even whose existence) must be determined by some computation. In this research, concerns are associated with identifying and extracting computations, rather than

\(^2\)An action noun is based on a verb.
identifying objects, for example, in the requirement “If the PIN is incorrect, the card is rejected.” The expression “...the PIN is incorrect...” must by some means be determined by a computation to verify the PIN. Hence, the expression is associated with a verification computation.

A predicate phrase denotes operations that return true or false values, such as checking status or state (e.g. isEnabled, isValid). A control structure phrase, on the other hand, denotes execution flow such as if..then..else, while, iterate, loop, after, selection. Some constraints identified from requirements can also determine control structures. See Section 5.3.2.3.

5.3.2 Identifying Component-based Constructs of the X-MAN Component Model

As discussed in Section 3, the key semantic concepts in the X-MAN component model are computation [LLW06], control [LLW06] and data [LT06, LT07]. In brief, computation means data transformation or function evaluation, whereby values or functions are computed and variables may be updated. Control means the flow of execution between components that encapsulate pieces of computations. Thus, the result of a piece of control invoking a computation is a piece of behaviour. Data, on the other hand, may be used as a constraint to control or to store useful values. In the current semantics of the X-MAN component model, it is assumed that data always follows control.

5.3.2.1 Identifying Keywords that Denote Computations

A computation can be defined as any data transformation process which may take any data as input, and perform some functions and outputs information in order to achieve a specific objective. Although data transformations can involve data
access operations i.e., input/output operations, the element that is of interest excludes the data access or manual operations. The smallest unit of computation is a single data transformation. A computation may consist of a single data transformation or a set of related data transformations (processes or methods). A computation unit is specified by its name.

<table>
<thead>
<tr>
<th>Category</th>
<th>Denotes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transformation</td>
<td>Computation(data transformation)</td>
<td>changePrice, search, update</td>
</tr>
<tr>
<td>State</td>
<td>Internal state of components (attribute values of components)</td>
<td>keep, remain</td>
</tr>
<tr>
<td>Event</td>
<td>Events that can trigger computation</td>
<td>press, enter, select</td>
</tr>
<tr>
<td>Action</td>
<td>Computations (data transformation)</td>
<td>identifier, validation</td>
</tr>
<tr>
<td>Descriptive Expression</td>
<td>May denote implicit computations</td>
<td>&quot;..change amount is displayed..&quot;</td>
</tr>
<tr>
<td>Predicate</td>
<td>Computations-evaluation operations that can be true/false</td>
<td>isNormal, isGreen</td>
</tr>
</tbody>
</table>

Fig. 5.5: Extracting computations.

A verb may express computation. However, not all verbs can be considered as computations. By referring to the definition of computations, candidates of computations are selectively identified based on (1) verbs that belong to one of these categories: Data Transformation, State or Event; (2) nouns that are Action nouns; or (3) phrases that are Descriptive Expressions or Predicates. The computation extraction category can be summarised in Fig. 5.5. Each element of the computation category is further analysed. In addition, examples are provided to motivate and demonstrate the identification process.

From the literature, existing categories of keyword extraction that are relevant to computations are adopted and adapted, according to what needs to be extracted. The first category is Data Transformation verbs. A Data Transformation verb denotes function evaluation, which takes data as input, and performs some processes and outputs data, in order to achieve a specific objective. Any database transaction can also be considered as data transformation, e.g. search,
update. This category is adopted from Action [SHE89, RP92]. A data transformation in general excludes manual operations, i.e. physical human activities, for example arrive, hand over, press, leave.

A few examples are provided in order to demonstrate how computations can be identified using the extracted keywords from the text based on a set of heuristics. In the element extraction process, the main aim of an analyst’s task is to use this set of heuristics to guide him in identifying computations as defined in the X-MAN component model (see Chapter 3). Note that not all verbs are computations. Thus, some of the verbs that cannot be categorised as computations have to be manually filtered.

Summary → 1 Identifying Computations from Text

HR-1A Computations can be identified using textual analysis based on (1) category of verbs: (explicit) data transformation; (implicit) state and event; (2) action nouns and (3) phrases: descriptive descriptions and predicates.

The following examples demonstrate how computations can be extracted based on the heuristic HR-1A.

Example 1

R: There is a single control button available for the use of the oven. If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, energise the power-tube for one minute).

Analysis. Based on HR-1A, a number of verbs can be identified. However, the verbs that are of interest are verbs that fall into either one of the three categories that have been laid out i.e. Data Transformation, State and Event. From this text, energise or cooking computations are identified.

An essential task here is to decide which category the identified verb belongs to; if the verb belongs to the first category, that is the Data Transformation,
a computation can be explicitly identified. On the other hand, if the identified verb belongs to either one of the other two verb categories, the corresponding computations are implied from implicit computations.

Example 2

R: If the PIN is incorrect, the card is ejected.

**Analysis.** For this statement, the relevant verb is *eject*, which can be categorised as a *Data Transformation* verb. Any interaction between system and hardware devices is considered as data transformation because processes are required in order to perform such interactions. In addition to this, the descriptive expression "is incorrect" may also imply computation. Thus, a computation to evaluate the PIN, i.e. authenticate computation, is needed.

Example 3

R: A user can withdraw up to £200 in units of £20 (the account may not be overdrawn).

**Analysis.** Here, a *withdraw* *Data Transformation* verb is extracted (assuming that the denomination of cash units is handled by the withdraw computation). In addition, the descriptive expression "may not be overdrawn" implies a data transformation to check if the amount is overdrawn.

Whenever there are interactions between the user and the system, the system shall provide the corresponding computations to handle these interactions. The aforementioned interactions will be mapped to the corresponding computations from the system's point of view.

**Summary → 2  Identifying Computations from User Interaction**

**HR-1B** Computations can be implied from any interaction between users and systems or hardware devices.
The following examples demonstrate how computations can be identified based on interactions between users and systems.

**Example 1**

R: When the user presses the ON button, the power tube will be activated.

**Analysis.** In general, most of the relevant words to be extracted will be identified using the POS tag set. In addition to the technique, interactions between users and systems should also be analysed. In this example, the user interacts with the system by pressing the ON button. In particular, here, the press verb is identified as an interaction and not based on the category of verbs. Hence, a computation must be provided to respond to this interaction. In this example, another activate verb is set as the computation. The decision is based on heuristic HR-1 from *Data Transformation* verb category.

**Example 2**

R: From the main menu, the clerk can choose one of the following options: Rent a tape, Return tapes.

**Analysis.** Based on HR-1B, the verb choose is identified as an interaction between the user and the system. The corresponding options for the choice are rent tape and return tape computations. Hence, here, two computations i.e. rent tape and return tape, are extracted.

**Example 3**

R: The Store Manager selects a product item and changes its sale price.

**Analysis.** For this requirement, although two verbs, i.e. select and change, are extracted, for each verb, the respective function evaluation that it corresponds to need to be considered. If there is no processing involved, then the candidate
computation is excluded. In this case, the change computation, which belongs to the Data Transformation verb category, is selected. This computation is anticipated to involve some processing of the item’s price.

Example 4

R: The Printer writes the receipt and the Cashier hands it out to the Customer.

Analysis. In the second example, although the writes and hands out verbs are identified, verbs that are physically performed by humans and which do not involve any data transformation are excluded. Thus, only the write computation, which belongs to the Data Transformation verb category, is selected.

Example 5

R: If the button is punched while the oven is cooking, it will cause the oven to cook for an extra minute.

Analysis. The following verbs are identified (1) punch as an Event verb; (2) cooking/cook as a Data transformation; (3) cause as an Event verb. Obviously, cooking can be mapped as a computation based on the category of verbs. The punch is actually an event that triggers cooking, hence no additional computation is needed here.

The second category in Fig. 5.5 is the State verb, which is adopted from State [SHE89, RP92]. A State verb denotes computations that realise states, i.e. change the data that belongs to components. Candidates of states can be identified from verbs (that can be extracted by the POS tagger, i.e. past simple, past participles, present participle) and adjectives [SHE89] that may imply system states.
Example 6

[UC2-1] The considered Cash Desk is in normal mode and has just finished a sale which matches the condition of an express checkout sale.

**Analysis.** Here, *normal* is an adjective of Cash Desk that means *normal* is a data item that belongs to the Cash Desk. It is implied that there must be a computation that handles the relevant data transformation.

The third category in Fig. 5.5 is the *Event* verb, which is adopted from *Emergence* [RP92]; it denotes an event that triggers computations. Any triggering events must be associated with its corresponding notifications, i.e. the respective computations. These computations are identified based on any interaction between users and systems, or hardware devices. Hence, whenever an *Event* verb consisting of interactions between the users and the systems is identified, the system shall provide the corresponding computations to handle those interactions.

Example 7

[UC2-2A] The Cashier presses the button Disable Express Mode. The colour of the Light Display is changed from green into black colour.

**Analysis.** The term **presses** is an event that triggers the change of Light Display from green into black. Here, this *Event* verb is addressed as denoting the computation to be dealt with i.e. change the light colour.

The fourth category in Fig. 5.5 is the *Action* noun, which is adopted from Action [SHE89]. An *Action* noun implies data transformation provided by a component, e.g. authentication, registration, initialisation.

Example 8

[UC1-R5b] In order to initiate card payment the Cashier presses the button
5.3. IDENTIFYING COMPONENT-BASED ELEMENTS FROM NLR

Card Payment at the Cash Box.

i. The Cashier receives the credit card from the Customer and pulls it through the Card Reader.

ii. The Customer enters his PIN using the keyboard of the card reader and waits for validation.

**Analysis.** In this case, clearly pressing a button (i.e. an *Event* verb) indicates interaction between the user and the system interface. Hence, in this requirement, when the system receives notification that the card payment button is pressed, the corresponding computation that deals with such interaction needs to be assigned. For this requirement, the relevant verbs are (1) *payment* and (2) *validation*, which both come from the *Action* noun category; (3) *enters*, which is a *Data transformation* verb (can be renamed into *readPIN* so that the computation is modelled from the system’s perspective).

So far, the first four categories in the table in Fig. 5.5 have been used to identify explicit computations, i.e. computations that are explicitly identifiable from the requirements. These computations correspond to keywords extracted by the POS tagger. However, POS tagging alone cannot uncover all the computations that are required. The main reason is that the requirements may not specify explicitly some of the intended computations. Furthermore, the functional requirements are written from the user’s point-of-view and not from the developer’s point-of-view. Thus, apart from identifying explicit computations, the *Descriptive Expression* and *Predicate* [Abb83] categories are used to guide the identification of implicit computations. These form the last two categories in Fig. 5.5.

A *Descriptive Expression* phrase, e.g. “...the change amount...”, may denote a computation to calculate the change amount. Abbott[Abb83] specifies that a *Descriptive Expression* describes a possible object whose identity (and possibly even whose existence) must be determined by some computation.
in this study, a *Descriptive Expression* is used to identify implicit computations from phrases.

**Example 9**

[UC6-R2] A report which informs about the delivery mean times is generated.

**Analysis.** The expression "... informs about the delivery mean times..." must somehow be determined by a computation to calculate the mean times of a delivery, hence it is concluded that the expression is associated with a calculate mean time computation.

A *Predicate* phrase denotes operations that can return true or false, such as checking status or state (e.g., `isInNormalMode, isBlack`).

**Example 10**

[UC1-11] If the Inventory is not available, the system caches sale.

**Analysis.** In this requirement, the "... is not available..." phrase that may denote computation to check the availability of the Inventory is identified. Hence, the `checkAvailability` computation is provided.

To conclude, based on this guidance from the computation category, candidates for computation units are identified. In order to assist an analyst to extract and identify component-based constructs, the *Extractor* tool highlights the extracted verbs, and identifies the verb categories to extract computations (see Section 5.5). Following this, he or she can also highlight nouns, and identify any relevant action nouns for candidates of computations.
5.3. IDENTIFYING COMPONENT-BASED ELEMENTS FROM NLR

5.3.2.2 Identifying Conceptual Components

Normally, computations are identified and assigned to the respective components that provide the services. Nonetheless, in a few exceptional cases, some words (e.g. action nouns, hardware devices, other sub-systems) in the requirements statement can directly imply a component. A conceptual component here signifies the identification of computation units that are derived from a component instead of identification of computations from the part-of-speech (POS) technique. By looking at the words, the analyst deciphers the information that reflect components; hence, computations are implied for such components.

When a requirement does not explicitly indicate a specific computation, but the analyst comprehends that there is a need to provide its corresponding computation to get the required result, this conceptual component can be applied. This is relevant whenever a component stated in the text without any explicit computation. If a candidate component is found without any explicit computation, this is classified as a conceptual component. For this purpose, the following heuristic depends solely on the noun extraction or action noun. Based on this justification, computations from the relevant extracted nouns can be implied. For the same reason, the decision is not formed on any type of syntactic structure of the statement.

Summary → 3 Identifying Conceptual Components

HR-2 Conceptual components can be identified directly from action nouns (based on textual analysis), or implied from hardware devices or any sub-systems.

The following example demonstrate how a candidate for conceptual component can be identified from the text.
Example 1

R: After each transaction, the ATM will display and print a receipt containing the transaction information.

Analysis. The transaction is an action noun, which can be analysed as a conceptual component because at this point the exact computations are not explicitly stated. Nonetheless, the analyst anticipates that computations are required to process such transactions.

5.3.2.3 Identifying Keywords that Denote Control

After computations in each requirement are identified, the following step is to look for control. A control such as if...then...else, while, iterate, loop, selection denotes execution flow, i.e. sequential, branching, looping [Abb83]. In identifying control, the same strategy as in identifying computation is followed, in which control is identified from: (1) explicit control from the extraction of the POS tagging process, (2) pre-defined control terms (Fig. 5.6), and (3) implicit control that may imply execution flow. The following examples introduce and motivate each of the mentioned categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Denotes/Implies</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preposition</td>
<td>Ordering</td>
<td>before, after, then, from</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Ordering</td>
<td>and, once</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Ordering</td>
<td>using, based</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Selection/condition</td>
<td>if, or</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Selection</td>
<td>branching, options, otherwise, choices, either alternatives, else, choose</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Looping</td>
<td>while</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Looping</td>
<td>loop, repeat, repetition</td>
</tr>
<tr>
<td>Preposition</td>
<td>Looping</td>
<td>until</td>
</tr>
</tbody>
</table>

Fig. 5.6: Extracting control.
The first category is *Preposition*. The prepositions of interest are prepositions of time that imply ordering (e.g. after, then, before), and prepositions that may imply origin of movement or direction\(^3\) (i.e. from, of). Prepositions that are not based on time, for instance prepositions of place (position and direction) (e.g. in, at, under, on, below) are of no interest in this research.

**Example 1**

R: The Enterprise Server is not available: The request is queued until the Enterprise Server is available and then is sent again.

**Analysis.** The preposition *‘then’* here denotes an execution order while the preposition *‘until’* implies repetition.

**Example 2**

R: The formatted file will then be counted.

**Analysis.** Here, the preposition *‘then’* denotes an execution order.

**Example 3**

R: Furthermore the corresponding Light Display (LD) is switched from black into green to indicate the Cash Desk’s express mode.

**Analysis.** The preposition *‘from’* here also denotes an ordering execution.

The *Conjunction* category can be broken down into three sub-categories, i.e. conjunctions that may imply ordering, selection (branching), and repetition (looping). According to Berry and Kamsties [BK00], *‘and’* denotes (1) concurrency of events or actions (2) conditions to be met (3) temporal order of events or actions or (4) enumerations which may not imply any ordering. Although concurrency

\(^3\)http://www.eslcafe.com/grammar/prepositions09.html.
is also an important aspect, as the current X-MAN component model supports single-threaded execution, ‘and’ that denotes concurrency is not handled.

Example 1

R: After each transaction, the ATM will display and print a receipt containing the transaction information.

Analysis. The ‘after’ preposition from Fig. 5.6 is a preposition that implies ordering execution.

Example 2

R: The System caches each sale and writes them into the inventory.

Analysis. Here, an ‘and’ conjunction that explicitly shows an ordering execution from caching sale transaction to update the inventory, is identified.

As discussed earlier, the conjunction ‘and’ does not always imply sequential ordering; it may suggest concurrency processes instead. Normally, whenever in conflict, a domain expert or an analyst decides whether the ‘and’ conjunction denotes ordering or not.

Example 3

R: Cash and also card payment is allowed and the Customer is allowed to buy as many goods as he likes.

Analysis. The ‘and’ conjunction here does not indicate sequential ordering, but indicates both payment methods, i.e. cash and card payment are available. In such a case, the analyst decides whether both payment methods are relevant or a choice has to be made between these methods. If the method payment is an option, thus, this ‘and’ conjunction denotes a selection case instead.
Example 4

R: The library system has two types of main users which are student and staff.

Analysis. The conjunction ‘and’ here means both student and staff instead of showing any kind of ordering. This type of statement is not considered as a functional requirement because it does not contain any behaviour.

*Control terms* denote or imply predefined execution flow which is not derived from conjunctions and prepositions, e.g. using, based, branching, selection, loop, repeat, otherwise, alternatives etc. These terms are selectively identified and are set as control terms. See Fig. 5.7. Nonetheless, we do not claim that these are the only control terms which can be set. Based on the experiences of executing case studies and examples, the list contains sufficient terms that can assist in identifying additional control keywords, in addition to the ones discussed in the other categories of the control extraction.

<table>
<thead>
<tr>
<th>Control Terms</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>branch, branches, branching, branched select, selection, selects, selected option, options choice, choices choose, chosen, chooses using, use, uses alternative, alternatives</td>
</tr>
<tr>
<td>Repetition</td>
<td>repeat, repetition, repeats, repeated iterate, iteration, re-iterate loop, loops, looping, looped</td>
</tr>
</tbody>
</table>

Fig. 5.7: List of Control Terms.

Examples 1 & 2

R: A student can be an undergraduate or a graduate. R: A course is taught by one or more lecturers.

Analysis. Even though the conjunction ‘or’ can be extracted from both of
these statements, neither of the conjunctions mean selection, but rather represent relationships information e.g. inheritance, is-a instead. In other modelling elements, such as Entity Relationship Diagram (database modelling) and Class Diagram (object-oriented modelling), such relationships are relevant. In contrast, in this work, only conjunctions that denote control are of interest to us.

Example 3

R: The Cashier enters the item identifier. The system displays the description and price. Otherwise, the product item is rejected.

Analysis. The keyword otherwise is identified. This term explicitly shows a branching execution from the identify item computation. In this example, there is no explicit control identified between identify item computation and display the information. However, implicitly, such an ordering is required from identify item to display the information computations. In this case, the analyst decides on the implicit control between more than one computation. This is sensible, and can be justified from the way the requirements are written.

Example 4

R: Using the item identifier the System presents the corresponding product description, price, and running total. The steps are repeated until all items are registered.

Analysis. The keyword repeated explicitly shows a looping execution for the identify item computation.

From an English language structure, control based on explicit prepositions from the text can be identified and inferred. So far, the identification of control is performed based on explicit control either from the POS tagging extraction (of conjunctions or prepositions) or from the predefined control terms. However, an
implicit control may be recognised from the requirements ordering, i.e. the way
the requirements are written.

Apart from this, implicit control may also be identified from descriptive ex-
pression.

**Example 1**

R: The maximum of items per sale is reduced to 8 and only paying by cash is
allowed.

**Analysis.** Besides the ‘and’ conjunction that denotes ordering execution,
this statement provides constraints that are useful for a looping structure, i.e. to
repeat eight times while identifying item transactions. Here, there are no explicit
keywords that lead us to extract a looping control structure. Nonetheless, such
a loop can be specified.

**Summary → 4 Identifying Control**

| HR-3 | Control can be (1) explicitly identified using control structure in textual analysis i.e. from conjunctions or prepositions; (2) identified using specific terms that semantically represent control flow; or (3) implied from preposition in text. |

**Issues With Control Keywords Extraction**

There may also be cases where data transformations might have been identified
without any extracted control from a single requirement. To resolve this issue, the
analyst can either (1) assume that the ordering of execution follows the way the
requirements are written i.e. sequentially; or (2) clarify the execution order with
the domain expert or the clients. The default assumption is that each statement
is written in order, even without any explicit indicator.

One of the possibilities that may contribute to such a scenario is when a
statement describes solely a single computation or a few computations that can
be provided by the same component. In effect, only a single component will be identified (without any connector). However, control execution is modelled between components and not in between computations. If more than one computation is identified and provided by the same component, only the selected computations to be exposed at the interface level will be specified. Accordingly, the invocation between these computations are treated as an internal execution. Thus, no explicit control is needed.

Example 1

R: A user enters card. A user enters PIN.

Analysis. Here, there is no clear indication of any ordering. However, based on the domain, it is assumed that the aforementioned statement is executed first.

In short, when there is no control identified, the alternatives are (1) if only a single component (or computation) is identified, the default assumption is that it is sequentially executed (with the previous requirement); or (2) need to clarify the execution order with a domain expert(s) or clients.

5.3.2.4 Identifying Keywords that Denote Data

As not all nouns are relevant or meaningful to be applied in the attempt to map keywords from requirements to component elements based on the X-MAN component model, the purposes of data in this work are laid out. The main purpose of data is to store any values that will be used by components. This includes storing constant and initialisation values [LT07]. Data that are used as constraints to be checked during condition or looping evaluation also need to be addressed. Ideally, we sought to extract keywords that denote relevant data based on the X-MAN component model. However, the only means that the POS tagger can be applied here is by extracting all the nouns from requirements.
Based on the result of the noun extraction, the following step is to filter only the applicable data based on values that are useful for computations, constraints for branching mechanism, or handling data dependencies between computations. This includes storing constant and initialisation values [LT07]. The following examples show the data extraction process.

**Summary → 5 Identifying Data**

**HR-4** Data can be identified and implied from the extracted nouns. The data are used for storing constant, initialisation values, constraints and selection values for branching purposes.

The succeeding examples exhibit how computations can be extracted based on the heuristic HR-4.

**Example 1**

Consider the following requirement that shows useful data for the identified computation:

R: The system displays the description and price.

**Analysis.** Here, description and price are extracted as relevant data to be used for the display computation.

**Example 2**

Consider the following requirement that shows data useful for a branching mechanism:

R: The Store Manager selects a product item and changes its sale price.

**Analysis.** In this requirement, apart from the product item and price as relevant data to be used for the change price computation, the Store Manager must also be verified prior to the invocation of the computation. Hence, data for
the Store Manager account should also be provided. This data will be used as validation data in order to verify its authorisation level.

Example 3

Consider the following requirement that demonstrates identification of data as a constraint:

R: If the entered amount of an incoming product is larger than the amount accounted in the Inventory, the input is rejected.

Analysis. In this statement, the amount is identified as a constraint on the amount accounted (useful for computation rather than for constraint) in the Inventory.

Examples 4 & 5

Consider the following requirement that shows useful data without implicit computation:

R: Each flight contains information of a flight code, departure airport, a destination airport, a date, departure time and arrival time.

R: A booking is for a particular flight and holds information about the ticket type, the passenger, the class and the payment details.

Analysis. Both of the above examples describe data elements or entities to store the corresponding values i.e. flight and booking, but contain no data transformation.

5.4 Example

The element extraction heuristics will be demonstrated using a simple yet complete example. Larger sets of requirements with the complete application of the
5.4. EXAMPLE

entire approach will be presented in Appendices C.

5.4.1 A simplified Automated Teller Machine (ATM) system

The complete requirements for ATM\(^4\) are as follows:

**R1** The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).

**R2** A customer must be able to make a cash withdrawal from the linked account. Approval must be obtained from the bank before cash is dispensed.

**R3** A customer must be able to deposit cash to the linked account that can be inserted into the cash slot. Approval must be obtained from the bank before physically accepting the cash.

**R4** A customer must be able to make a transfer of money between any two accounts originated from the linked account.

**R5** A customer must be able to make a balance enquiry of the linked account.

**R6** If the customer fails to be authenticated, the card will be rejected.

**R7** After each transaction, the ATM will display and print a receipt containing the transaction information.

In this example, keywords to be extracted for each requirement are analysed. First, computation identification based on HR-1A, HR-1B or HR-2 are performed. This is followed by control identification based on HR-3. Finally, data identification is done based on HR-4. The following Fig. 5.8 summarises the extraction heuristics as a guidance in analysing each requirement.

\(^4\) http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Requirements.html
CHAPTER 5. EXTRACTING ELEMENTS OF CB SYSTEMS

Summary of Element Extraction Heuristics

[H-1A] Identifying computations from textual analysis
Computations can be identified using textual analysis based on (1) category of
verbs: (explicit) data transformation; (implicit) state and event; (2) action nouns
and (3) phrases: descriptive descriptions and predicate.

[H-1B] Identifying computations from user interaction
Computations can be implied from any interaction between users and systems
or hardware devices.

[HR-2] Identifying conceptual component
Conceptual components can identified directly from action nouns (based on
textual analysis), or implied from hardware devices or a system.

[HR-3] Identifying control
Control can be (1) explicitly identified using control structure in textual analysis
i.e. from conjunctions or prepositions, (2) identified using specific terms that
semantically represent control flow, (3) implied from preposition in text.

[HR-4] Identifying data
Data can be identified and implied from the extracted nouns.

Fig. 5.8: Summary of the Element Extraction Heuristics.

Requirement-1

R: The ATM will service one customer at a time. A customer will be required
to insert an ATM card and enter a personal identification number (PIN).

Analysis. For R1, based on user interactions with the system (see HR-1B),
which are by inserting card and entering PIN, are identified. The system to be
built must provide computations to read the inserted card information and the
entered PIN. Hence, these two interactions imply computations. The next step
is to identify any keywords for control. Using textual analysis, based on HR-3,
the ‘and’ conjunction denotes an ordering execution. Finally, the requirement is
scanned through for data identification using HR-4. In this particular statement,
based on the extracted nouns, placeholders for customer, card and PIN data are
extracted.

Requirement-2

R: A customer must be able to make a cash withdrawal from the linked ac-
count. Approval must be obtained from the bank before cash is dispensed.

Analysis. Consider R2. Based on HR-1A, the computation verbs i.e. ap-
prove, dispense cash, and the action noun cash withdrawal as computations
Table 5.1: Summary of the Extracted Keywords from R1.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>Data Transformation (DT) (Fig. 5.5)</td>
<td>specify based on system's point of view (HR-1A)</td>
</tr>
<tr>
<td>enter</td>
<td>DT (Fig. 5.5)</td>
<td>specify based on system's point of view (HR-1A)</td>
</tr>
<tr>
<td>and</td>
<td>Conjunction (ordering) (Fig. 5.6)</td>
<td>textual analysis (HR-3)</td>
</tr>
<tr>
<td>customer, card, PIN</td>
<td>Data</td>
<td>based on textual analysis (HR-4)</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of the Extracted Keywords from R2.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>approve</td>
<td>DT (Fig. 5.5)</td>
<td>based on HR-1A</td>
</tr>
<tr>
<td>withdrawal</td>
<td>Action-noun (Fig. 5.5)</td>
<td>based on HR-1A</td>
</tr>
<tr>
<td>dispense</td>
<td>DT (Fig. 5.5)</td>
<td>based on HR-1A</td>
</tr>
<tr>
<td>before</td>
<td>Preposition (ordering) (Fig. 5.6)</td>
<td>based on HR-3</td>
</tr>
<tr>
<td>account, bank</td>
<td>Data</td>
<td>Store values (HR-4)</td>
</tr>
<tr>
<td>approval</td>
<td>Data (constraint)</td>
<td>data to be passed as constraint (HR-4)</td>
</tr>
</tbody>
</table>

(see Fig. 5.5) are identified. The subsequent step is to find keywords for control. The main task is to identify any guidance that may lead to either one of the three options of control i.e. ordering, selection or branching basis. In this statement, the preposition ‘before’ can be used as a hint to denote ordering (see Fig. 5.6). The result of the approve computation has to be checked before the withdraw and dispense cash computations are to be performed. Hence, the approval status has to be stored as data.
CHAPTER 5. EXTRACTING ELEMENTS OF CB SYSTEMS

Requirement-3

R: A customer must be able to deposit cash to the linked account that can be inserted into the cash slot. Approval must be obtained from the bank before physically accepting the cash.

Analysis. For R3, first, look for computations or candidate components, then find hints for control, and finally search for data (if any). Here, the computation verbs deposit, accept cash and approve as computations are identified. However, since the same approve computation has already been identified in the former requirement, for this step, this particular computation can be skipped. The preposition ‘before’ implies some kind of ordering. This is based on HR-3. Next, look for any relevant data. Clearly, an approval status needs to be checked before a user is allowed to perform the deposit transaction.

Table 5.3: Summary of the Extracted Keywords from R3.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/nouns/phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit</td>
<td>DT (See Fig. 5.5)</td>
<td>based on verb HR-1A</td>
</tr>
<tr>
<td>accept</td>
<td>DT (See Fig. 5.5)</td>
<td>specify based system interaction (HR-1B)</td>
</tr>
<tr>
<td>before</td>
<td>Preposition (See Fig. 5.6)</td>
<td>based on HR-3</td>
</tr>
<tr>
<td>approval</td>
<td>Data (constraint)</td>
<td>data to be passed (HR-4)</td>
</tr>
</tbody>
</table>

Requirement-4

R: A customer must be able to make a transfer of money between any two accounts originated from the linked account.

Analysis. From the fourth requirement, another ATM transaction that can transfer money from the linked account, is identified. In addition, the linked account from the descriptive phrase implies that an authentication process is also required in order to perform the money transfer process. The ‘from’ preposition
that implies ordering execution and the account as data are also extracted. As a result of having implicit authentication computation, an additional approval status is required as a constraint data. The summary of extraction is listed in Table 5.4.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer money</td>
<td>DT (Fig. 5.5)</td>
<td>based on verb HR-1A</td>
</tr>
<tr>
<td>linked account from</td>
<td>Descriptive expression (Fig. 5.5)</td>
<td>implicit that an authentication is needed</td>
</tr>
<tr>
<td></td>
<td>Preposition (ordering) (Fig. 5.6)</td>
<td>based on textual analysis (HR-3)</td>
</tr>
<tr>
<td>transfer account</td>
<td>Data</td>
<td>based on HR-4</td>
</tr>
<tr>
<td>approval status</td>
<td>Data (constraint)</td>
<td>data to be passed (HR-4)</td>
</tr>
</tbody>
</table>

**Requirement-5**

R: A customer must be able to make a balance enquiry of the linked account.

**Analysis.** From the fifth requirement, a balance enquiry computation is extracted. With the same justification, authentication computation is required for verifying the linked account. Subsequently, the preposition ‘of’ denotes origin, which indicates ordering execution. Eventually, customer data is also identified.

**Requirement-6**

R: If the customer fails to be authenticated, the card will be rejected.

**Analysis.** For the sixth requirement, based on the analysis (see Table 5.6), the approve and reject card computations are identified here. However, as approve computation has already been identified in the earlier increment, the
CHAPTER 5. EXTRACTING ELEMENTS OF CB SYSTEMS

Table 5.5: Summary of Extracted Keywords from R5.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>enquiry</td>
<td>Action noun (Fig. 5.5)</td>
<td>based on noun (HR-1A)</td>
</tr>
<tr>
<td>linked account</td>
<td>DT (Fig. 5.5)</td>
<td>implies that authentication is needed (HR-1A)</td>
</tr>
<tr>
<td>of</td>
<td>Preposition (Fig. 5.6)</td>
<td>based on textual analysis (HR-3)</td>
</tr>
<tr>
<td>customer Data</td>
<td></td>
<td>has already been documented (HR-4)</td>
</tr>
</tbody>
</table>

only computation that needs to be included is the reject card computation. For control, the ‘if’ conjunction that implies a selection case is extracted. Here, if the customer fails to be authenticated, only then will the card be rejected. Both card and customer information data have also been documented in the earlier increments.

Table 5.6: Summary of the Extracted Keywords from R6.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticate</td>
<td>DT (Fig. 5.5)</td>
<td>based on HR-1A</td>
</tr>
<tr>
<td>reject</td>
<td>DT (Fig. 5.5)</td>
<td>based on HR-1A</td>
</tr>
<tr>
<td>if</td>
<td>Conjunction (Selection) (Fig. 5.6)</td>
<td>based on HR-3</td>
</tr>
<tr>
<td>customer Data</td>
<td></td>
<td>has already been documented (HR-4)</td>
</tr>
</tbody>
</table>

Requirement-7

R: After each transaction, the ATM will display and print a receipt containing the transaction information.

Analysis. Consider the seventh requirement. As depicted in Table 5.7, the approve and reject card computations are identified. Nevertheless, since approve computation has already been identified in the earlier increment, the only
computation that is required is the reject card computation. For control, the ‘if’ conjunction that implies a selection case is extracted. Here, if the customer fails to be authenticated, only then will the card be rejected. Both card and customer information data have also been documented earlier.

<table>
<thead>
<tr>
<th>Extracted keywords</th>
<th>Category of verbs/ nouns/ phrases</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction</td>
<td>Action noun (See Fig. 5.5)</td>
<td>Conceptual component HR-2</td>
</tr>
<tr>
<td>display</td>
<td>DT (See Fig. 5.5)</td>
<td>display information HR-1B</td>
</tr>
<tr>
<td>print</td>
<td>DT (See Fig. 5.5)</td>
<td>print transaction HR-1B</td>
</tr>
<tr>
<td>after</td>
<td>Preposition (see Fig. 5.6)</td>
<td>implies ordering (HR-3)</td>
</tr>
<tr>
<td>and</td>
<td>Conjunction (ordering) (Fig. 5.6)</td>
<td>based on textual analysis (HR-3)</td>
</tr>
<tr>
<td>receipt, transaction</td>
<td>Data</td>
<td>Store values for transaction, receipt (HR-4)</td>
</tr>
</tbody>
</table>

5.5 Implementation of the Extractor Tool

A tool to assist an analyst during the element extraction process is developed. Fig. 5.9 depicts how the Extractor tool works. Initially, a requirement will be used as an input to the tool. Each word will be syntactically tagged using a pre-defined selected POS tagger set.

In addition, the tool includes built-in heuristics which allow filtering of irrelevant architectural related elements. Thus, a user (analyst) shall be able to highlight words according to verb or noun features. The main task of the user is to identify computations or candidate components, control and data. The following step is to map those extracted keywords to architectural elements as defined in the X-MAN component model. This part will be discussed in Chapter 6.
5.5.1 Application of a Part-Of-Speech Tagger

The X-MAN component model defines components, connectors and their composition mechanism. Based on this notion, the elements to be identified from text are derived. The main elements are computations, control and data. In order to syntactically extract these elements, the Brown Corpus Part-Of-Speech (POS) tagger, which can be used to tag the three main categories, is adopted. The categories are (1) verb, (2) noun and (3) control.

The Brown Corpus tagset contains 176 types of tags. In this work, irrelevant tags are removed and the most relevant tags that can be used to extract elements based on the X-MAN component model, are selected. Each of the categories is derived based on the following tag set\(^5\).

Note that this work does not provide a detailed discussion on each of the selected tag set. A summary of POS tagger and its usage in textual analysis is provided in Appendix A. The main intention is to apply this tag set to each statement. For verb extraction, five types of tagsets are selected: (1) verb, present (2) verb, past tense, (3) verb, present participle or gerund, (4) verb, past participle

\(^5\)http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html
5.5. IMPLEMENTATION OF THE EXTRACTOR TOOL

and (5) verb, present tense, 3rd person singular. These tag sets are selected from the Brown Corpus tagset.

<table>
<thead>
<tr>
<th>TAG</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VB</td>
<td>find, calculate, display, assign, formulate, plan, organise, enter, select ...</td>
</tr>
<tr>
<td>VBD</td>
<td>produced, took, found, added, charged, voted, delivered, received, scheduled ...</td>
</tr>
<tr>
<td>VBG</td>
<td>improving, purchasing, enabling, pricing, keeping, getting, entering, making, setting ...</td>
</tr>
<tr>
<td>VBN</td>
<td>conducted, charged, studied, revised, operated, accepted, combined, experienced ...</td>
</tr>
<tr>
<td>VBZ</td>
<td>deserves, believes, receives, takes, starts, permits, expects, thinks, eliminates, sets ...</td>
</tr>
</tbody>
</table>

Fig. 5.10: POS Tagset for Verb Extraction.

On the other hand, the adopted five types of tag set for noun extraction are (1) noun, singular, common, (2) noun, singular, common, genitive, (3) noun, plural, common, (4) noun, plural, common, genitive, (5) noun, singular, proper, (6) noun, singular, proper, genitive, (7) noun, plural, proper, (8) noun, plural, proper, genitive, (9) noun, singular, adverbial, (10) noun, plural, adverbial, and (11) noun, singular, adverbial, genitive.

For the control keyword extraction, three main groups of POS tagsets are applied, these being (1) conjunctions i.e. conjunction, coordinating (CC) and conjunction, subordinating (CS), (2) determiners or quantifiers i.e. determiner/pronoun (DT), singular Determiner/pronoun (DTI), singular or plural determiner, pronoun or double conjunction (DTX), determiner/pronoun or pre-quantifier (ABN), determiner/pronoun, double conjunction or pre-quantifier (ABX), determiner/pronoun or post-determiner (AP), determiner/pronoun, post-determiner, genitive (AP$) and (3) prepositions (IN).
CHAPTER 5. EXTRACTING ELEMENTS OF CB SYSTEMS

Fig. 5.11: POS Tagset for Noun Extraction.

<table>
<thead>
<tr>
<th>TAG</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN  - noun, singular, common</td>
<td>airport, appointment, payment, light, microwave, system, receipt, transaction ...</td>
</tr>
<tr>
<td>NN$ - noun, singular, common, genitive</td>
<td>season's, world's, player's, night's, chapter's, department's ...</td>
</tr>
<tr>
<td>NNS - noun, plural, common</td>
<td>sessions, reports, choices, years, areas, adjustments, sales, details, fees ...</td>
</tr>
<tr>
<td>NNS$ - noun, plural, common, genitive</td>
<td>taxpayers', children's, members', women's ...</td>
</tr>
<tr>
<td>NP - noun, singular, proper</td>
<td>September, October, London, System ABC ...</td>
</tr>
<tr>
<td>NPS - noun, singular, proper, genitive</td>
<td>Michaeil's, John's, Smithy's ...</td>
</tr>
<tr>
<td>NPS - noun, plural, proper</td>
<td>British, French, Congresses ...</td>
</tr>
<tr>
<td>NPS$ - noun, plural, proper, genitive</td>
<td>Republicans', Yanks', Russians' ...</td>
</tr>
</tbody>
</table>

These control elements extracted from the POS tagger may be used to identify explicit control. Among them, some may be used to denote an ordering or non-ordering flow of execution. For instance, the prepositions `before' or `after' may imply an ordering execution, as discussed in HR-3. In addition to the POS tagset for control extraction, specific pre-defined terms (not from the POS tagset) that can be used to identify control, as discussed in HR-3, are defined (see Fig. 5.6).
5.5. IMPLEMENTATION OF THE EXTRACTOR TOOL

5.5.2 Element Extraction Using the Extractor Tool

A simple tool for editing and analysing the keywords extracted from requirements has been implemented based on the flow as depicted in Fig. 5.9.

Fig. 5.13 shows a screen shot of the tool. Initially, a requirement will be an input to the tool. Each word will be syntactically tagged using a pre-defined selected POS tagset. In addition, the tool includes built-in heuristics which allow filtering of irrelevant words. For instance, articles (e.g. ‘the’, ‘a’, ‘an’) will not be extracted. Moreover, a user can highlight words according to verb, noun or control features (Fig. 5.13). The tool thus helps by suggesting keywords that may denote control, computation or data based on the POS tagset and some predefined rules (see Fig. 5.13).

![Extractor Tool](image)

Fig. 5.13: The Extractor Tool.

Using the same ATM example, the keywords extraction process is demonstrated using the Extractor tool.

Requirement-1

R: The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).

The extraction. For R1, the tool extracts [1] verb, [2] noun, and [3] control structure. For verb, service, required, insert, enter are extracted. In addition to this, for noun, ATM, customer, time, customer, ATM, card,
identification, number, PIN are extracted. Finally, for control, at, and are extracted.

These elements are extracted based on the selected POS tagsets and the pre-defined terms for control identification. The POS extraction is restricted to syntactic tagging only. Later, the analyst or designer may manually select the relevant extracted elements based on the component model's semantics of computation, control and data. At this stage, they need to acquire knowledge concerning what are computation, control and data as defined in the adopted component model. Instead of analysing the whole requirement statement, they only need to be concerned with these three categories of elements i.e. based on verb, noun and control. Eventually, the verb, noun and data categories will be mapped to computations, data (or conceptual component) and control. The mapping of the extracted keywords to the elements of X-MAN component model will be presented in Chapter 6.

5.6 Issues and Discussion

In approaches that represent information in the requirements into abstraction modelling concepts, the entire set of requirements has to be analysed. In contrast, this element extraction process is confined to be applied to a single requirement at a time. Hence, such a strategy reduces the degree of the relevant abstraction process, which are typically subjected to the expertise of the systems analyst or designer. This section elaborates issues pertaining to the element extraction process.
5.6. ISSUES AND DISCUSSION

5.6.1 Requirements Problems

This element extraction process does not claim to solve problems in requirements such as ambiguities, incompleteness or inconsistencies. These issues should be resolved and clarified with the customer prior to the use of the proposed approach or during the extraction process. It is worth noting that the problems with the requirements are beyond the scope of this work, and it is still an open issue that is being researched in the Requirements Engineering (RE) community. Nonetheless, the problems with the requirements may contribute to the issues in the later stages of this approach.

5.6.2 Dealing with Implicit Requirements

It is acknowledged in the literature that some requirements may need to be derived or invented, as discussed by Goguen & Linde [GL93]; Goldin & Berry [GB97]. Nonetheless, the focus of this work is concerned with the extraction process from NLR, mapping of the extracted keywords, how the mapped elements are used to construct component-based system and not how the requirements should have been written or constructed. In this work, the original requirements are not pre-processed or changed into a pre-defined syntax. From the extraction process, the descriptive expression category is used to suggest implicit computations or control. At present, this is the only solution to address implicit requirements, whilst attempting to reduce the analyst’s assumptions during the extraction.

5.6.3 Issues on Computation Identification

There are cases where no computation can be identified from the requirements. The implication of not having any computation during the element extraction process is that no architecture (based on the X-MAN component model) can
be derived from the requirement. The possible causes of this issue are requirement statements, which only contain data elements or statements that contain verbs, which describe relationship information, e.g. contains, consists, inheritance. Although RE area suggests guidelines to produce concise and unambiguous requirements statements, in reality, there is no restriction as to how the NLR can be expressed. This research deals mainly with functional requirements. If requirements are stated or written without specifying their functionalities, computations could not be directly extracted from them. However, these data might be useful for other computations. If the data are potentially applicable, they are documented for future purposes.

5.6.4 Handling More Than One Control Extraction

Natural language has no precedence and associativity as in mathematics or programming concepts [BK00]. In programming, precedence rules are applied when dealing with more than one operator e.g. and, or, if, etc. To make this precedence rule straightforward, the use of brackets is essential. Nonetheless, in this research, concerns are focused on identifying control, which encapsulates computation execution and how this can be used to meaningfully and accurately represent the behaviour that satisfies a specific requirement. No rule for such a composition is provided. Nevertheless, the motivation of identifying more than one control is to accurately identify the relevant computations and to apply heuristics to identify composition connectors in later stages.

5.6.5 Limitations of the Elements Extraction

In the analysis of NLR, many other elements could also be identified than the defined keywords at present. The X-MAN component model is a control-driven
5.7. SUMMARY

and single-threaded execution model. As the premise of the extraction process is motivated by this model, hence the focus of extraction at present is constrained to computations, control and data.

The model can be refined to incorporate these additional elements, e.g. active components, data flow, etc., and indeed, different versions of the model are being constructed at the time this thesis is written in order to accommodate them. As the generic elements of the model are applied, hence only these elements are motivated as the basis of the extraction. With this constraint, the types of system that are mostly suitable for this approach to work are control-based and single-threaded execution systems. Apart from this, an equivalent work around or a temporary solution has to be considered.

5.7 Summary

This chapter has discussed the element extraction process and proposed the element extraction heuristics in the mission to identify and decipher information from the requirements. The extraction process is formed based on the X-MAN component model elements. We believe that as the X-MAN component model supports high cohesion and no coupling between computations, such an extraction process should be easy as compared to other component-based architecture such as ADL-like and object-oriented. These heuristics will assist the designer in extracting relevant keywords from the requirements and map these keywords to the elements of the X-MAN component model, which will be presented in Chapter 6.
Chapter 6

Mapping from Requirements to Architectural Elements

This research attempts at constructing component-based systems directly from NLRs. In order to achieve this, relationships (1) between candidates of architectural elements in the NLRs and the extracted keywords that denote architectural elements and (2) between the extracted keywords to the architectural constructs using the X-MAN component model, must be defined. Hence, the main aims of this chapter are to define the mapping process, to propose and justify design decisions with regards to the mappings of the possible candidates of architectural elements from requirements to the extracted keywords and from there to component-based systems elements.

6.1 The Mapping Process

In this section, we start by presenting an overview of the mapping process; and continue discussing the correspondences between (1) the elements from requirements and the extracted elements and (2) between the extracted elements and
6.1 THE MAPPING PROCESS

the XMAN architectural constructs.

6.1.1 Overview

Mapping and transformation have long been investigated in the domain of database [VMP03, JSPC02]. Model Driven Architecture (MDA) suffers from lack of consensus on terminologies, especially pertaining to mapping and transformation [LHBJ05]. In addition, mapping is defined as the application or execution of a mapping function in order to transform one model to another [MSUW04, JSPC02], and mapping function is defined as a collection of mapping rules that defines how particular mapping works [MSUW04].

On the other hand, transformation generates a target model from a source model [GGKH03]. In defining transformation, [MVGO6] differentiate between endogeneous and exogeneous transformation, which relates to expression used whether in the same language or not. Work such as [JSPC02] uses the mapping and transformation terms interchangeably.

According to [LHBJ05], the mapping and transformation terms should be explicitly distinguished. They added that mapping specification comprises the mapping definition of the correspondences between metamodels, whilst transformation contains the description to transform a model into another using a transformation language. The transformation can either be manual or automatic generation from a source model to a target model according to a transformation definition [LHBJ05]. Model transformation can be automated or manual (may need certain level of human intervention) [MVGO6]. A mapping specification could be used with many different transformation definitions [LHBJ05]. Apart from that, some propositions enabling the mapping specification have been used based on heuristics [RB01].
In this thesis, we use the term mapping as synonyms for correspondences between (1) the NLR and the extracted elements; and (2) the extracted elements from NLR and the XMAN elements. The overall mapping process can be exemplified in Fig. 6.1 and the mapping is denoted by the solid arrows in the figure.

![Fig. 6.1: Mapping Process](image)

As shown in Fig. 6.1, the mapping process focuses on the candidates of architectural elements in the NLR. These elements will be mapped to (1) keywords that denote computation; (2) keywords that denote control; and (3) keywords that denote data. These keywords will then be mapped to architectural elements. A system design architecture using XMAN semantics comprises three main elements i.e. components, connectors (composition connectors and adapters) and data. Discussion on the mapping definition is provided in Section 6.1.2.

A simplified version of a UML class diagram representing the relevant XMAN constructs that are of interest in this research is presented in Fig. 6.2. It is important to note that not all XMAN constructs depicted in Fig. 6.2, are mapped from NLR. That is, not including the derived classes. A derived class such as the Interface class can be derived from computations of the component and no attempt has been made to discover the Interface from NLR. The main elements that are of interest are computations, control and data. For computations and
control, the representing classes are direct, which belong to Computations of Component and Connector classes. On the other hand, data can be mapped as constraint for Guard, condition for Loop adapters, and condition for SEL connector.

![UML Class Diagram](image)

Fig. 6.2: UML Class Diagram Representation

### 6.1.2 The Mapping Definition

The definitions of these architectural elements for the purpose of mapping from NLR to the XMAN model are presented below.

**Definition 1.** Requirement (RQ) elements are all possible XMAN architectural elements stated in the NLR.

These RQ elements are candidate of architectural elements which will be mapped to keywords that denote computations, control or data. Nonetheless, not necessary all the identified RQ elements will be mapped to those keywords. There might be cases where there exists more than one similar terms that can
represent the same keywords. For instance, we might identify more than one control terms. In the end, we might need to decide which control term can be applied for the mapping to keywords that denote control.

**Definition 2.** Extracted elements (EE) contain (1) keywords that denote computation; (2) keywords that denote control; and (3) keywords that denote data. In each requirement, these three categories of elements should be identified to be mapped to the XMAN elements.

The extracted elements will be associated with XMAN architectural elements.

**Definition 3.** The XMAN elements (XE) consist of (1) component that encapsulates computations; (2) connector; and (3) data. A component can have more than one computations; hence, the interface exposes the component’s provided services (computations).

Computations are services provided by components. Hence, the extracted keywords that denote computations will be mapped to computations of components. There might also be cases where some of the intended behaviours of the extracted elements can be achieved by the same component.

A connector can be defined based on its name, type, arity and scheme. A connector can be of type composition or adaptation connector. For explanation of these connectors, see Section 3.5.2. An arity attribute relates to the number of component and connector instances, which will be connected to the connector. Meanwhile, the adaptation connector is a unary connector, hence only a single component can be connected to it. The scheme attribute defines the execution order. If the connector type is a \texttt{SEQ}, the scheme will define the ordering execution sequence.

In XMAN, data is relevant for checking constraints, to store values for checking
6.1. **THE MAPPING PROCESS**

condition or default value. At the current state of work, the focus is more on components and connectors. At the point of this research, we restrict the use of data for handling these cases.

To complete the mapping process definition, it is essential to define mapping function to relate (1) the candidates of XMAN architectural elements from NLR and (2) the extracted elements from NLR to the XMAN architectural elements. In this work, these correspondences are represented using the semantic net, which is used to signify occurrences of entities in the database domain [CB05, p. 346] and [SKS10, p. 269].

![Fig. 6.3: Mapping Relationship](image)

Mapping cardinalities show the number of entities to which another entity can be associated via a relationship set [SKS10, p. 269]. In Fig. 6.3, the mapping cardinalities describe binary relationship\(^1\) sets between (1) all the possible candidates of XMAN architectural elements from each wording in the requirement statements and the extracted elements and (2) the extracted elements (EE) and the XMAN elements (XE).

The elements in RQ (denoted by \(r_q\)) are possible candidates of the XMAN architectural elements. Meanwhile, the elements in EE (denoted by \(e_e\)) are the elements in EE (denoted by \(e_e\)) are the

---

\(^1\)relationship type between entities in one entity type and entities in another entity type
keywords that denote computations, control and data. The elements in XE (denoted by xe) are the components, connectors and data as represented in Fig. 6.2. The relationship from RQ elements to EE elements is one-to-many, in which an entity in RQ is associated with at most one entity in EE. Whilst, the other relationship from EE to XE is also one-to-many, whereby an entity in EE can be associated with many entities in XE. However, not all EE elements will eventually be mapped to XMAN architectural constructs. In the end, the user decides and filters as to which computation, control or data keywords that are useful to be applied.

Based on Fig. 6.3, we can say that an RQ element can be associated to many elements in EE. As an example, if a term can be represented as candidate for a computation, hence, the term can be mapped into a keyword that denotes a computation. Because RQ represents all possible architectural elements, human intervention is required to arbitrate the design decisions in this process. Hence, each RQ element may be mapped to a number of computations, control or data.

In another scenario, by referring to the same figure, an EE element can be mapped to many elements in XE. For instance, when a computation A is extracted from NLR, it is possible that it can be mapped to many components. This is subjected to which component that provides the required services to be used in the composition as to satisfy the required behaviors. This scenario will also be addressed in Algorithm-3, in which each of the extracted keywords will be mapped to computations, control or data, respectively. In another example, if we identify a control term (from RQ element) that denotes ordering e.g. after, it can be mapped to SEQ in the repository. There might be more than one composition connectors to represent sequencers in the repository as this is in deployment phase.
6.2 Mapping Function of the Requirement (RQ) Elements to the Extracted Elements (EE)

This section summarises the discussion on the mapping of the NLR elements into the corresponding desired modelling elements. Research on the use of heuristics to aid the identification and extraction of RQ elements from NLR has been investigated. See Section 5.3 and Section 5.3.1 for the discussion. Most object-based mapping approaches rely to a large extent on identifying nouns and verbs because the object-oriented computational model supports only operands and operators [Boy99]; object-oriented software development [Boo86] uses the rule that nouns correspond to objects or classes, whereas verbs correspond to messages between objects. In this section, a mapping function from the RQ elements to the EE elements will be presented.

6.2.1 Mapping RQ Elements to Keywords that Denote Computations

Given that the intention of this research is to map requirements to architectures in the X-MAN component model, elements in requirements that correspond to the key semantic concepts in the model need to be identified, and hence, words (in requirements) that represent these concepts also have to be extracted. As presented in Chapter 3, one of the key semantic concepts in the X-MAN component model is computations. A computation means data transformation or function evaluation, in which values or functions are computed and variables may be updated.

The main elements to identify candidate for data transformation can be identified from verbs. Verbs generally refer to actions, events and processes [Jac82].
A computation verb, e.g. withdraw, denotes a data transformation, which takes data as input, performs some function evaluation and outputs data, in order to achieve a specific objective. In general, data transformations can involve simple data access operations i.e. input/output operations; however, these kind of computation verbs are not used to denote such data transformations. Only database operations as computations, e.g. search, updates are considered. Verbs that are physically performed by humans such as walking, eating, etc., are excluded as data transformation.

<table>
<thead>
<tr>
<th>Category</th>
<th>Denotes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Transformation</td>
<td>Computation(data transformation)</td>
<td>changePrice, search, update</td>
</tr>
<tr>
<td>State</td>
<td>Internal state of components</td>
<td>keep, remain</td>
</tr>
<tr>
<td>Event</td>
<td>Events that can trigger computation</td>
<td>press, enter, select</td>
</tr>
<tr>
<td>Action</td>
<td>Computations (data transformation)</td>
<td>identifier, validation</td>
</tr>
<tr>
<td>Descriptive Expression</td>
<td>May denote implicit computations</td>
<td>isNormal, isGreen</td>
</tr>
<tr>
<td>Predicate</td>
<td>Computations-evaluation operations</td>
<td>change amount is displayed</td>
</tr>
</tbody>
</table>

Fig. 6.4: Category of RQ Elements that Denote Computations

A state verb, e.g. keep, maintain, cooking denotes computations that realise states, i.e. change the attribute values of components. However, not all state verbs can be mapped to a computation. Normally, states which are of interest are ones that can be associated with computations. In order to achieve this kind of state, computations need to be performed. This means that the system should provide corresponding computations. States such as idle which may not be doing anything, do not imply any computations.

An event verb, e.g. press, denotes an event that can trigger computations. This Event verb can indirectly imply computations that should be provided by the system. Because the requirements are unstructured, such hints can be a guidance in determining candidate computations.

An action noun\(^2\), e.g. registration, denotes data transformation provided

\(^2\)An action noun is based on a verb.
by a component such as authentication and initialisation processes. In the extraction process, which is supported by the Extractor tool (see Section 5.5), elements of the component-based system are suggested based on verbs, nouns or phrases using a textual analysis technique. By excluding the computation noun category, the extraction process might not be comprehensive in covering the extraction of all candidate computations.

In addition to the verb and noun categories, a descriptive expression phrase, e.g. the earlier date, may also denote computations. In this research, concerns are associated with identifying and extracting candidates for computations, rather than identifying objects, for example, in the requirement “If the PIN is incorrect, the card is rejected.” The expression “..the PIN is incorrect..” must by some means be determined by a computation to verify the PIN. Hence, the expression is associated with a verification computation.

A predicate phrase may also denote operations that return true or false values, such as checking status or state (e.g. isEnabled, isValid).

In addition to the above guidances, in a few exceptional cases, some words (e.g. action nouns, hardware devices, other sub-systems) in the requirements statement can also imply a component. When a requirement does not explicitly indicate a specific computation, but the analyst comprehends that there is a need to provide its corresponding computation to get the required result, this conceptual component can be applied. This is relevant whenever a component stated in the text without any explicit computation. This is when we identify a conceptual component. By analysing the RQ element, the analyst may comprehend that there is a need to provide its corresponding computation to get the required result, this conceptual component can be applied. This is relevant whenever a component stated in the text without any explicit computation.
6.2.2 Mapping RQ Elements to Keywords that Denote Control

The other main architectural element as defined in XMAN is control. In identifying control, the same strategy as in identifying computation is followed, in which control can be identified from: (1) explicit control from the extraction of the POS tagging process, (2) pre-defined control terms (Fig. 5.6), and (3) implicit control that may imply execution flow. Section 5.3.2.3 provides examples to motivate the defined categories.

Fig. 6.5 lists possible RQ elements with examples, which may be mapped to keywords that denote control. The possible RQ elements are grouped into three categories i.e. (1) based on ordering; (2) based on selection; and (3) based on looping (repetition). For instance, if we come across the term before, we can identify that as a candidate of keywords that denote control.

<table>
<thead>
<tr>
<th>Category</th>
<th>Denotes/Implies</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preposition</td>
<td>Ordering</td>
<td>before, after, then, from</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Ordering</td>
<td>and, once</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Ordering</td>
<td>using, based</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Selection/condition</td>
<td>if, or</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Selection</td>
<td>branching, options, otherwise, choices, either alternatives, else, choose</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Looping</td>
<td>while</td>
</tr>
<tr>
<td>Control Terms</td>
<td>Looping</td>
<td>loop, repeat, repetition</td>
</tr>
<tr>
<td>Preposition</td>
<td>Looping</td>
<td>until</td>
</tr>
</tbody>
</table>

Fig. 6.5: Category of RQ Elements that Denote Control

The first category is Preposition. The prepositions of interest are prepositions of time that imply ordering (e.g. after, then, before), and prepositions that may imply origin of movement or direction\(^3\) (i.e. from, of). Prepositions that are not based on time, for instance prepositions of place (position and direction) (e.g. in, at, under, on, below) are of no interest in this research.

\(^3\)http://www.eslcafe.com/grammar/prepositions09.html
The *Conjunction* category can be broken down into three sub-categories, i.e. conjunctions that may imply ordering, selection (branching), and repetition (looping). The term ‘and’ denotes (1) concurrency of events or actions (2) conditions to be met (3) temporal order of events or actions or (4) enumerations which may not imply any ordering\[BK00\]. Although concurrency is also an important aspect, as the current X-MAN component model supports single-threaded execution, ‘and’ that denotes concurrency is not handled.

*Control terms* denote or imply predefined execution flow which is not derived from conjunctions and prepositions, e.g. using, based, branching, selection, loop, repeat, otherwise, alternatives etc. These terms are selectively identified and are set as control terms. See Fig. 5.7. Nonetheless, we do not claim that these are the only control terms which can be set. Based on the experiences of executing case studies and examples, the list contains sufficient terms that can assist in identifying additional control keywords, in addition to the ones discussed in the other categories of the control extraction.

From an English language structure, control based on explicit prepositions from the text can be identified and inferred. So far, the identification of control is performed based on explicit control either from the POS tagging extraction (of conjunctions or prepositions) or from the predefined control terms. However, an implicit control may be recognised from the requirements ordering, i.e. the way the requirements are written.

Apart from this, implicit control may also be identified from descriptive expression. In this case, a *control structure* phrase is a candidate for keywords that denote execution flow such as if..then..else, while, iterate, loop, after, selection. Some constraints identified from requirements can also determine control structures.
6.2.3 Mapping RQ Elements to Keywords that Denote Data

Not all nouns are relevant or meaningful to be mapped as keywords that denote data. The main purpose of data is to store any values that will be used by computations. This includes storing constant and initialisation values [LT07]. Data that are used as constraints to be checked during condition or looping evaluation also need to be addressed. Ideally, we sought to extract keywords that denote relevant data based on the X-MAN component model. However, the only means that the POS tagger can be applied here is by extracting all the nouns from requirements.

Based on the result of the noun extraction, the following step is to filter only the applicable RQ elements based on values that are useful for computations, constraints for branching mechanism, or handling data dependencies between computations. This includes storing constant and initialisation values [LT07]. See Section 5.3.2.4 for examples that motivate the design decision to identify keywords that denote data.

6.3 Mapping Function of the Extracted Elements to XMAN Architectural Elements

In this section, the mapping function from the extracted elements to the XMAN architectural elements will be discussed. It is important to note that not all of the extracted elements will be mapped to the XMAN elements. Our focus is to represent the intended behaviors as stated in the requirement. For instance, if the same or similar terms are extracted, the user will decide as to which element that is going to be applied.
6.3.1 Mapping Keywords that Denote Computations to Components

A component is not explicitly stated in NLR because a component is defined by its properties. That is why, from NLR, we extract the required computations or the behaviours from the RQ elements, and not the component directly. These computations are derived from text based on (1) verb that are data transformations, state or event; (2) action noun; and (3) phrases, that are from descriptive expression and predicate stated in the requirements. See Section 5.3.2.1 on Identifying Keywords that Denote Computations and Fig. 5.5 for computation extraction.

As presented in Chapter 3, the X-MAN component model defines components that encapsulate computations. These computations are services provided by components. In the X-MAN model, components have only the provided interface and have no required interface. It is important to emphasise that in the X-MAN semantics, components do not communicate directly with each other; instead, communication is coordinated by the composition connectors that encapsulate control. The following discussion demonstrates how the extracted keywords are mapped into the component’s computations.

Based on the extracted computation keywords from Fig. 5.5, these keywords are mapped to computations that are encapsulated or provided by components. These keywords are assigned to the respective logical components. The following Examples 1–3 demonstrate this process.

Example 1

R: After each transaction, the ATM will display and print a receipt containing the transaction information.
**Analysis.** For this requirement, the extracted display and print computations from the element extraction process (using the Extractor tool) are mapped to computations in the component-based construct. In this case, both of the computations are relevant, hence they are selected. These computations can be assigned to be handled separately by Printer (PRT) and Display components.

**Example 2**

R: The customer has his ABC card with him. The account number of a customer is read with the bar code reader from the clerk to retrieve the rental transaction record.

**Analysis.** The extracted read card and retrieve computations from the element extraction process are mapped to computations in the component-based construct. Now, these computations are mapped to CardReader (CR) and Rental Transaction Record (RTR) components respectively.

**Example 3**

R: When the clerk presses an order-complete option key (defined by the system) this rental is complete and the video inventory file is updated.

**Analysis.** The extracted update data transformation from the element extraction process is mapped to an update computation in the component-based construct. In this case, the computation is assigned so as to be provided by the RTR component.

### 6.3.1.1 Design Decisions on Combining Computations or Separating Computations in Different Components

There are cases where more than one computation is extracted; a decision has to be made as to whether the computations can be grouped together in a single
component or simply to set each computation to be dealt with by each component. By separating computations into distinct components, accordingly (based on its functionalities) this strategy increases the probability of finding the respective components from a repository.

As a guidance, if computations are based on device operations (e.g. printer, display devices), all the related computations can be grouped as services provided by a single component. Otherwise, separate the computations based on its functionalities. Referring to Example 1, alternatively both the extracted computations can be mapped so as to be handled by the same component i.e. Print (PRT) component because both computations are possibly dealt with by a same device.

Summary → 1 Mapping Computation Keywords to Component’s Computations

| HM-1 | The extracted computation keywords are mapped to computations. These computations are assigned to the respective logical components that provide the required services. |

6.3.2 Mapping Keywords that Denote Control to Connectors

A connector as defined in XMAN encapsulates control, which is used for communication and coordination of components. Control can be derived from text based on (1) explicit control from the extraction of the POS tagging process, (2) pre-defined control terms (see Fig. 5.7), and (3) implicit control that may imply execution flow. See Section 5.3.2.3 Identifying Keywords that Denote Control and Fig. 5.6 for control extraction. As presented in Chapter 3, the X-MAN component model defines three types of generic composition connectors, namely the
PIPE, SEQ and SEL.

In the case where more than one control are identified from requirements, the user decides as to which control can be applied to represent the execution of the intended behaviours between the mapped components. This section further demonstrates how the extracted control keywords are mapped to these connectors.

6.3.2.1 Mapping to a PIPE or SEQ connector

In general, an explicit inference for deciding a composition connector is based on the keywords that denote control (see Section 5.3.2.3). These control keywords are extracted from conjunctions (e.g. and), prepositions (time preposition e.g.: before, after, then) and control terms (e.g. using) that imply ordering. Nonetheless, not all occurrences of these control keywords and control terms can be directly mapped to composition connectors. This is why human judgment is still required to justify the selection of the extracted elements.

For instance, the ‘and’ conjunction does not always imply a sequential ordering, and may suggest concurrency process instead. According to Berry and Kamsties [BK00], the ‘and’ conjunction may denote (1) concurrency of events or actions; (2) conditions to be met; (3) temporal order of events or actions; or (4) enumerations which may not imply any ordering. In this work, the only case that is of interest is the third case which denotes ordering of events or actions. Although concurrency is also important, as the current X-MAN component model supports single-threaded execution, this work only incorporates the ‘and’ conjunction that denotes ordering.

In addition, there are also control terms, which are not directly extracted from NLR using the POS tagger, can be used to identify ordering execution. The examples of these terms are as listed in the ordering control terms in Fig. 5.6.

Once an ordering is determined as being required for the execution of the
composition, the control keywords are mapped to either a PIPE or a SEQ. The main factor to be considered when deciding either to adopt a PIPE or a SEQ is whether there is any data dependency or not. The decision is based on whether any data needs to be passed between the connected components. If the composition requires data, the control is mapped to a PIPE connector; otherwise, the control is mapped to a SEQ connector.

### 6.3.2.2 Mapping to a Selector Connector

In deciding the mapping of control keywords to composition connectors, the execution category, namely ordering, selection or repetition, needs to be identified. The ordering execution cases are mapped to either a PIPE or a SEQ, whilst the selection control keywords are mapped to a SEL.

In principle, an ‘if’ statement is mapped to a selection statement. However, not all cases with an ‘if’ statement can be treated as a branching solution. Although an ‘if’ statement clearly implies a branching statement, according to the X-MAN semantics, an ‘if’ statement cannot be simply mapped to a SEL. A SEL as defined in the X-MAN component model is an n-ary composition connector that composes two or more components and defines the branching control that chooses one of the connecting components to be invoked based on a specific condition [Eli08, p. 99]. According to this definition, the control keywords can only be mapped to a SEL connector when a branching control between two or more connecting components are identified. Otherwise, the ‘if’ statement is mapped to a guard adapter that evaluates the condition (data constraint).

When an ‘if’ statement with only a single identified computation is extracted, the if control is mapped to a guard that evaluates the constraint instead of mapping the if to a SEL. During the later stage, if a pattern for branching can be detected, only then can the design be considered to be refactored into a
SEL. The refactoring tasks will be discussed in Chapter 7. In addition, the ‘or’ conjunction denotes either one or the other but not both [BK00]. This term can also be used to map to a SEL with the constraint that more than one component has been identified.

So far, the extraction category is focused on the explicit control keywords that can be identified from NLR using the POS tagger. Apart from this, the defined control terms, which are not directly extracted from the POS tagger, can also be used to suggest the selection-based execution. The examples of these terms are as listed in Fig. 5.6.

Summary → 2  Mapping Control Keywords to Composition Connectors

**HM-2** Control keywords are mapped to composition connectors based on ordering or selection (branching) execution.

### 6.3.2.3 Handling More Than One Composition Connector

It is possible to have more than one control execution for each requirement. The main challenge here is how to correctly map the extracted control keywords so as to satisfy a particular requirement. When more than one control is identified, it is important that the execution orders between computations are maintained. In order to handle any complex behaviours in a single requirement, it is inevitable to distinguish the execution order.

**Example 4**

Consider an example from VSS requirement.\(^4\)

R: Only managers can delete customers or video record. Manager enters an account number of video or customer. The entered video or account record

\(^4\)Taken from [LSB98]
will be deleted.

**Analysis.** Looking at the above statement, based on the POS tagger extraction rule, the **delete customer** and **delete video** data transformations are identified. Apart from this, based on a descriptive expression that is "only managers", it is assumed that a computation to authenticate the manager’s status must be provided. The ‘or’ conjunction here can be mapped to a **SEL** or a guard. However, since more than one computation has already been identified, this control can directly be mapped to a **SEL**. The result of the validation status will be passed to the **delete customer** and **delete video** computations. Hence, this control is mapped to a **PIPE** because data is passed from the **AUT** component to the **CustomerRecord (CR)** and **VideoRecord (VR)** components.

For documentation purposes, the execution flow is loosely specified using the following notation: \( CC([G] \rightarrow C_1, \ldots, C_n) \) where \( CC \) represents the generic composition connector namely **PIPE**, **SEQ** or **SEL**; \( C_1 \ldots C_n \) represent the composition units that are the components; and the \([G]\) represents the optional guard adapter that contains the constraint data.

In Example 4, the execution flow that is mapped from the ‘or’ conjunction is between the **CR** and **DR** components. This flow can be specified as **SEL(CR, VR)**. In addition, another execution flow is between the **AUT** and both of the computations. This flow can be specified as **PIPE(AUT, SEL(CR, VR))**. However, an additional guard is needed to check the constraint on the manager’s status. Hence, the guard is added into the specification as:

\[ \text{PIPE}(\text{AUT}, \text{Guard(status)} \rightarrow \text{SEL(CR, VR)}) \]
6.3.2.4 Design Decisions on Composition Connectors

During design, it is possible to produce many design alternatives. In the R2 of the ATM example (see Fig. 6.7), a composition is built between Withdraw (WD) and Cash Dispenser (CD) components so that the resulting composite component can be further composed with the Authentication (AUT) component. The design of the composition can either be as illustrated in Case-1 (Fig. 6.7a) or Case-2 (Fig. 6.7b). For this particular requirement, all three identified components can be composed using a single PIPE connector based on the extracted preposition keywords as the basis for the execution order. Having a design which contains all the three components linked with a single PIPE, as illustrated in Case-1 (Fig. 6.7a), is not a design error.

The disadvantages of such a design are that (1) further composition is limited to composition from the top level connector\(^5\) or at the available composition points; and (2) the design is not logically grouped together. Although the design can correctly represent the current behaviours from the mapped elements, for future composition, it lacks a meaningful logical structure that can lead to effective design.

Alternatively, consider having two separate PIPEs as a result of the composition, as shown in Fig. 6.7b. This decision is particularly useful if a particular composition is reusable in a different part of the system architecture. Accordingly, when part of a composition is anticipated as being reusable, it is more convenient to have a separate branch of composition.

As a consequence of incorporating an additional PIPE, observe that a new hierarchy level is created. By having such a design, more transactions are allowed to be directly linked (composed) to the open composition points (at both levels).

\(^5\)This type of composition can also be referred to as normal composition.
in the future.

Consider that in the subsequent requirement, a new bank transaction is introduced, i.e. *deposit*, which will also need to be authenticated prior to its execution. With the design in Case-2 (Fig. 6.7b), the additional bank transaction can be directly composed using the same connector (i.e. the second level connector, PIPE), rather than having no other choice of a correct composition point to satisfy the required behaviours. Case-3 (Fig. 6.7c) illustrates the result of additional behaviour composition from the previous architecture as designed in Case-2 (Fig. 6.7b). Such a composition is allowed with the features of incremental composition, which will be separately discussed in Chapter 4.

If the initial composition is designed as shown in Fig. 6.7a, one of the possible design alternatives can be as shown as in Case-4 (Fig. 6.7d). However, the disadvantage of such a composition is that every time a new composition is added, as long as there is data dependency in the former composition, they must be
propagated, along with the new composition. This can lead to ineffective design. Nevertheless, this issue can be solved using refactoring. Refactoring issues and tasks will be presented in Chapter 7.

As a summary, in deciding how to compose or deciding the number of connectors, particularly when more than two components are being dealt with, the system designer can anticipate whether any of the components might be useful to other components. If this feature can be anticipated in advance, the components should be composed using separate connectors.

### 6.3.3 Mapping Control Keywords to Adapters

So far, the extracted keywords and control terms that denote ordering from conjunction and preposition, and the keywords and control terms that denote selection for multiple components have already been mapped to either PIPE or SEQ. Apart from this, control keywords that denote the selection of multiple components are mapped to SEL, whilst the control keywords that denote selection or repetition but are only applicable to a unary component are mapped to adapters. This section discusses how these control keywords are mapped to adapters.

#### 6.3.3.1 Mapping Control Keywords to Guard Adapters

As described in Chapter 3, there are two generic types of adapter, namely guard and loop adapters. Guards are used to check any constraints or conditions before an invocation to the computation is made. The conditions determine whether the execution flow continues with the computation invocation or returns back.

![Fig. 6.8: An Example of a Guard Adapter.](image-url)
In an example shown in Fig. 6.8, the system execution flows from approve computation, which is provided by the AUT component and the result of authentication, namely the approval status, will be passed to the connecting component. Upon receiving a valid approval status from AUT component, only then will the withdrawal operation be executed. When a single component is identified, the checking can only be done by a guard and not a composition connector. When data (constraint) are required, a PIPE is required to pass the data to be checked at the constituent branches.

**Example 5**

R: If the customer fails to be authenticated, the card will be rejected.

**Analysis.** Since only a single component is identified here that is the AUT, a guard should be applied here instead of a SEL.

**Example 6**

Consider the following VSS requirement.

R: If past-due amounts are owed, they can be paid at this time; or the clerk can select the "order-complete" key which updates the rental with the return date and calculates past-due fees. The rental transaction record is updated.

**Analysis.** In this statement, the update (must update Rental Transaction Record (RTR) as well) and the calculate fee computations can be directly identified. Both of these computations can be logically grouped into a single component i.e. Overdue (OVD) component (see Fig. 6.9).
Since the overdue process is not an independent process and it is significantly relevant to the return tape process, for modelling purposes, the Return (RTN) component is added in the design. Based on the descriptive expression, a constraint is required to check whether the past-due amounts are owed or not. This feature can be realised by adding a guard that checks the constraint. In order to pass data between RTN and OVD components, a PIPE connector is added to compose them.

6.3.3.2 Mapping Control Keywords to Loop Adapters

Unlike the guard adapter, a loop adapter handles repetition cases. In repetition execution, constraints to repetition are checked and if the conditions are satisfied, the invocation of the respective computation will occur repeatedly until the conditions for the constraints have changed and hence the conditions are not fulfilled anymore. This will end the repetition and the control will return to its caller. Repetition may lead to uncertainties, and this explains why we need to ensure that the loop will eventually terminate safely. Counter loop, for instance, has a clear cut termination as compared to the condition loop which may lead to such problems.

Fig. 6.10 shows how a loop adapter can be modelled. In the Trading System (COCOME) [RRMP08] example, each item will be identified using a scanner and the item’s information will be displayed on the terminal screen. The identifying and displaying computations will be performed repeatedly until an acknowledgement is received. Here, the acknowledgement is the constraint to be checked for
6.3. MAPPING FUNCTION EE TO XE

the loop adapter.

Summary → 3  Mapping Control Keywords to Adapters

HM-3 Decision to map control keywords to adapters is based on selection (constraint) or repetition execution.

6.3.4 Mapping Keywords that Denote Data to XMAN Data

Data can be identified and implied from the extracted nouns. As stated earlier, data in XMAN semantics are only relevant for storing constant, initialisation values, constraints and selection values for branching. See Section 5.3.2.4 Identifying Keywords that Denote Data.

When a requirement contains only information on data elements such as database elements, optionally, these information can be specified as data of the component-based element. At this stage, more concern is focused on handling control and computations, instead of data. Nonetheless, data that are relevant to the mapping include constraints for PIPE, SEL and constraint data for branching mechanism or constraints for adapters (i.e., guards and loop).

Whenever a constraint data is identified and selected, accordingly, a control for evaluating the constraint has to be created. This strategy is to ensure that the constraint is properly addressed. This control is later mapped to a guard adapter.

Summary → 4  Mapping Data Keywords to Data Elements

HM-4 The extracted data keywords (constraints for PIPE, SEL and constraint data for branching mechanism or constraints for adapters) are mapped to data elements.
6.3.5 Discussion

The provided mapping functions are not direct mapping of the extracted elements to the architectural elements. Hence, human interventions are required to justify any design decisions. This is when the mapping heuristics are applicable.

The followings are the mapping heuristics relevant to be applied during the mapping process:

1. The extracted computation keywords are mapped to computations.
   - These computations are manually (by a human analyst) assigned to the corresponding logical components that provide the required services.

2. Control keywords are mapped to connectors based on ordering or selection mechanism.

3. Control keywords are mapped to adapters based on constraints on computation or repetitive execution.

4. The extracted data keywords used for constraints for connectors are mapped to data elements.

6.4 Issues and Discussion

This section addresses relevant issues and discussion during the mapping of the extracted elements from NLR into component-based systems constructs.

6.4.1 Mapping of More Than One Control Extraction

During the element extraction process, we may encounter cases where more than one control is extracted. In this case, the analyst needs to specify the computations with regard to the relevant control keywords. For example, in the seventh
requirement of the ATM example, two control keywords were extracted. Here, the 'and' conjunction is clearly for an execution between display and print computations, whilst the 'after' preposition is meant for implying ordering between transactions and both the computations. This strategy is required to identify correctly the relevant computations and their control execution; and then map these elements to their corresponding X-MAN constructs. In the end, no matter how many controls are being extracted, only a single partial architecture should be derived. This partial architecture will be composed with the incremented system architecture.

6.4.2 Handling Redundancies

The derived partial architecture should be designed based on the extracted elements. Some of these elements might be redundant. For example, in the ATM requirements, the hints that denote authentication process appeared more than once in separate requirements. Nonetheless, in such a case, the designer's aim is to represent the extracted elements as to satisfy the behaviours of each of the requirements. From another perspective, the redundancy issue is useful to detect a valid composition point when the partial architecture is composed to the system architecture. Without this information, the partial architecture may need to be deferred. The redundancies can later be addressed during the refactoring and the incremental composition tasks.

6.4.3 Limitations of the Mapping Step

When the mapping is performed, which is based on the extracted elements, we do not claim that there is a single and definite design solution. In this chapter,
although only one solution is provided for each of the mappings of a single requirement, it is not necessary that there is only one way to represent a design. The resulting designs are merely an attempt at satisfying the stated behaviours in the requirements. Hence, no effort has been put to analyse or compare the best possible design in this stage. Nonetheless, during refactoring, the restructuring of an architecture is motivated mainly to achieve simplification of a design or to enable further composition.

6.5 Summary

This chapter has presented a set of design decisions and heuristics grounded on the extracted keywords elaborated in Chapter 5. In addition to the heuristics, examples of the requirements mapping for each of the categories namely for the computations, control and data have demonstrated the applicability of the design heuristics. Once the extracted keywords are mapped into component-based elements, the following step is to model the elements into partial architectures and to incrementally build a working system architecture. Nonetheless, during incremental steps, the derived architecture may need to be re-structured in order to allow further composition or to simplify the design. Thus, the subsequent chapter exemplifies how such refactoring of the architecture can be realised.
Chapter 7

Architecture Refactoring

This chapter defines and justifies the architecture refactoring tasks to be applied in the approach to constructing component-based systems directly from NLR. These refactoring tasks are motivated by examples and further verified case-by-case to prove that the original behaviours are preserved, whilst changes to the architecture are carried out. The chapter starts by setting out a brief introduction to architecture refactoring and continues with the justifications of the refactoring rules. These refactoring rules are applicable during each of the incremental steps and also during the finalisation step.

7.1 Introduction

Refactoring, in general, is defined as “restructuring software by applying a series of refactorings without changing its observable behavior” [FBB+99]. The term ‘software’ here may include its internal structure, design, code or even any relevant software artifacts. In [MVG06], refactoring in model-driven development is viewed as a transformation process, which preserves the behaviour needs while the structure is modified. However, the general concept of refactoring that is
typically used in the literature is closely related to source code level rather than modelling level.

Despite the common use of refactoring in the literature, the concept has also been used in the context of architecture [Pas04]. In [MQR95], structural changes are made with the intention of making transformations between architectural styles (e.g. pipe-and-filter, client server, layering etc.).\(^1\) In this chapter, the discussion will be specific to architecture refactoring only, thus restricting its general definition to any other types of software artifacts.

The promising benefits of refactoring are (1) to reduce architectural design errors, and (2) to provide systematic reuse of design knowledge and proofs [MQR95, TB01]. Refactoring facilitates reusability and design proofs. Once the design is proven, we no longer need to re-evaluate its correctness and simply reuse the design knowledge. This way, cost of debugging and re-testing can also be reduced. Without refactoring, there may be duplication of efforts or unnecessary errors during implementation.

Having said this, the challenges here are to identify and define the design knowledge and provide case-by-case evidence of correctness between the original and resulting architecture that has been refactored. The derived architecture guarantees compositionality, meaning that it enables the composition to be further composed into a larger composition. In addition, the most important feature to be maintained during performing refactoring tasks is the preservation of behaviours.

The objectives of the architecture refactoring in this approach are (1) to modify the architecture structure, and (2) to improve the architecture design, while maintaining the incremented behaviour of the original architecture. For the aforementioned objective, during incremental steps, the architecture may need to be

\(^1\)In [MQR95], the architecture refinement term being used is considered as refactoring.
re-structured. A foreseen reason for this is when dealing with problems while composing the partial architecture into system architecture. To be more precise, this case happens when there is no possible or available composition point to be used. However, in order to preserve the behaviours that have been mapped in the former increments, refactorings are restricted to the rules that have already been justified and proved. Otherwise, the behaviour preservation feature is not guaranteed. These rules will be discussed in Section 7.3.

Meanwhile, the latter objective allows modifications of an architecture to improve the architectural design. For this objective, some of the architecture refactoring rules are introduced to improve the design in the context of its structure. An important premise behind this is to reduce the complexity of the derived architecture. For instance, identifying PIPE with guards pattern (with some constraints) can be mapped into a PIPE with a SEL. The resulting design helps to devise a more descriptive architecture and also reduces the use of adapters (guards). As a result, this task can assist in reducing the complexity of the derived architecture.

### 7.2 Refactoring Tasks

The architecture refactoring tasks can be categorised into (1) *vertical* and (2) *horizontal* refactorings. The *vertical* refactorings permit the hierarchy of connectors to be reduced. When the level of hierarchies is minimised, the level of complexity in the context of the architectural design could also be inherently reduced (see Fig. 7.1(a)). Meanwhile, the *horizontal* refactorings allow the expansion or shrinking of components or compositions. It is generally expected that the architecture keeps on expanding as increments are performed (see Fig. 7.1(b)).

The expansion of components or composition is possible without any addition
to hierarchy levels because of the *open composition* points. While composing during each increment, these open composition points are utilised. Indirectly, the effects of increasing additional hierarchy levels and design complexity can be significantly reduced. For instance, *homogeneous* connectors\(^2\) can be combined into a single connector. As a consequence of this particular refactoring task, unnecessary connector hierarchies could be abbreviated. Accordingly, the architectural design could also be simplified.

### 7.3 Architecture Refactoring Rules

This section provides the motivation and justification for all the refactoring rules applied in this approach. Without these justifications, the mission to preserve the incremented behaviours cannot be accomplished. These rules will be presented based on the refactoring tasks category, namely the *vertical* and *horizontal* refactoring. For each refactoring, the explanation covers the purposes of the rule, its analysis, example(s) and the anticipated issues. In the incremental approach, during composition, behaviour preservation is the most important aim to be achieved. During the formulation of the rules in defining architecture refactoring, the designer has to ensure this property is being addressed. In order to achieve this, two factors are required to be considered: (1) behaviours being

\(^2\)Connectors which are the instances of the same type of composition connector.
7.3. ARCHITECTURE REFACTORING RULES

represented; and (2) ordering of the behaviour composition. Whilst performing incremental construction, if both of those factors are satisfied, we consider the represented behaviours are preserved.

One plausible way to provide an argument for the refactorings is to use a customised Unified Modelling Diagram (UML) activity diagram [Stö04] so as to focus on the execution flow. The same approach has also been applied in [VL10]. The notation is that of a dotted rectangle (box) denotes composition. The rounded rectangle with label (e.g. A, B, C) represents the components. An arrow shows the control flow from one end to the other. In each of the included rules, activity diagrams will be added to represent the impact of the refactorings.

7.3.1 Vertical Refactoring

Three refactoring rules are specified under the vertical refactoring category. They are (1) flattening the hierarchy levels, (2) adding a lower-level hierarchy and (3) adding an upper-level hierarchy. In this study, the number of connectors is considered to be proportional to the architecture’s complexity level. Hence, our aim is to reduce the number of connectors whenever possible while maintaining behaviour preservation. However, architecture refactoring is required in this approach not only to reduce complexity, but also to improve the architectural design. This is due to the need to satisfy the stated behaviour in the requirement and also to produce a simplified and descriptive architectural design.

7.3.1.1 Flattening the hierarchy levels

By flattening the hierarchy levels, these levels are technically combined. Consequently, this strategy will inevitably lead to the reduction of the hierarchy levels. As the complexity of an architecture corresponds to the number of connectors
and its hierarchy levels, the complexity can be reduced in this context.

**Rule-1: Combining composition connectors**

The rule for combining composition connectors is only applicable to *homogeneous* connectors. Homogeneous connectors are connectors instantiated from the same connector type (e.g. PIPE, SEQ, SEL). For example, two SEQs can be combined into a single SEQ subject to some constraints (in this case, they must be of adjacent hierarchy levels). To use this rule, a designer scans through the architecture design and looks for the connectors in the architecture that can be vertically combined.

**Case-1: Homogeneous connector**

Consider the initial architecture in Fig. 7.2(a). If the architecture is composed of a new homogeneous connector e.g. a PIPE, the resulting composition can be depicted in Fig. 7.2(b). This refactoring rule generally suggests that the design can be simplified into a single connector (see Fig. 7.2(c)).
7.3. ARCHITECTURE REFACTORIZING RULES

The refactored composition can be represented as:

\[
Arch_{ori} = CC_1(C_A, C_B) \\
= CC_1(CC_2(C_A, C_B), C_C)
\]

\[
Arch_{target} = CC_{12}(C_A, C_B, C_C)
\]  \hspace{1cm} (7.1)

where \( CC_{1..n} \) = composition connector, \( C_{A..C} \) = component or composition, \( Arch \) = architecture; and \( CC_{1..n} \) must be homogeneous. In (7.1), the homogeneous connectors \( CC_1 \) and \( CC_2 \) are combined into a single connector.

The above rule can also be represented using a UML activity diagram. Fig 7.3 exemplifies the same PIPE execution as represented in Fig 7.2. In the original architecture in Fig. 7.2(a), a PIPE is used to compose A and B components. The execution of the PIPE can also be represented as in Fig 7.3(a). Whenever a new homogeneous connector, in this case, a PIPE is used to further compose AB with component C, a new PIPE is used as shown in Fig. 7.2(b). This is equivalent to the execution represented in Fig. 7.3(b). Whenever homogeneous connectors are used in composition, the connectors can always be combined into one connector construct. This is supported by the same features in Fig 7.3(b) and Fig 7.3(c).

---

Fig. 7.3: Representing execution flow for PIPE.
An important issue to be considered is how the exposed interfaces are being handled. In order to preserve the composed behaviour as a result of the former composition, more interfaces (in the form of method signatures) are added to the corresponding connectors every time a new composition is carried out. The newly added interfaces shall be propagated to all the consecutive upper hierarchy levels. The following Fig. 7.4 demonstrates the solution to this issue. In the original design (see Fig. 7.4(a)), the interface of the composition exposes the \( m_1m_2() \) computation. In the subsequent increment, for instance, as shown in Fig. 7.4(b), a new component is composed to the architecture. Now, the newly exposed interface has to be updated to accommodate and reflect the new addition. Once the homogeneous connectors are refactored, they are combined into a single connector as shown in Fig. 7.4(c).

Fig. 7.4: Handling interfaces

A significant benefit of refactoring homogeneous connectors is to simplify the design. Consider if a large number of requirements is being dealt with and if the composition is restricted to be performed with the top level connector only\(^3\), the outcome of this design decision may lead to an explosion of hierarchy levels. These unnecessary hierarchies can be reduced and thus simplify the resulting

\(^3\)This composition is also referred to as *normal* composition.
architecture. The aim of this work is to achieve the required behaviours, so as to
satisfy the mapped requirement, and finally, the required system. Hence, as long
as this aim is maintained and the benefits of these refactoring rules are gained,
it is necessary to apply them whenever required.

Example. The following is drawn from the WordCount [Sch06, p. 434] example.
In Fig. 7.5a, an input file is validated in ValidateFile (VF) component and
the result of the validation is piped to the Display Error (DE) (in the former
increment step) or Format File (FF) (in the later increment step) components
respectively. In this example, after an increment step, an architecture as in
Fig. 7.5b is produced. As depicted in the figure, the two homogeneous connectors,
i.e. the PIPEs, are refactored into a single PIPE.

This example can be represented as:

\[
Arch_{ori} = PIPE_1(C_{VF}, [\text{status}=\text{false}] \rightarrow C_{DE})
\]
\[
 = PIPE_1(PIPE_2(C_{VF}, [\text{status}=\text{false}] \rightarrow C_{DE}), [\text{status}=\text{true}] \rightarrow C_{FF})
\]
\[
Arch_{target} = PIPE_{12}(C_{VF}, [\text{status}=\text{false}] \rightarrow C_{DE}, [\text{status}=\text{true}] \rightarrow C_{FF})
\quad (7.2)
\]

where \( PIPE_{1..n} \) = heterogeneous composition connector, \( C_{A..C} \) = component or
composition, \( \text{Arch} = \text{architecture} \).

It is worth mentioning here that the resulting architecture in Fig. 7.5c can be further refactored using the PIPE with guards pattern, which will be discussed in the subsequent section.

**Case-2: Heterogeneous connector**

In contrast to homogeneous connectors, heterogeneous connectors consist of a different combination of connector types. This rule is only applicable to homogeneous connectors. The main reason is that any refactorings for the heterogeneous connectors are restricted because behaviour preservation is difficult to achieve.

**Summary → 1 Combining Homogeneous Composition Connectors**

**AR-1** Two or more relevant homogeneous composition connectors can be combined into a single connector.

### 7.3.1.2 Adding a Lower-level Hierarchy

As requirements are not expected to be in any specific order, the approach has to provide the means to be able to support bottom-up and also top-down composition mechanisms. In the XMAN model, composition is done bottom-up, whilst execution of the composition is done top-down [VL10]. However, whenever required, this rule shall be able to support the need to perform top-down composition, that is, decomposing parts of an architecture. This may be caused by any relevant behaviours that logically belong to an existing part of a composition. For this case, only heterogeneous connectors are considered. The main reason is that for any case of homogeneous connectors, rule AR-1 has already covered the case, hence can be directly applied.
7.3. ARCHITECTURE REFACTORIZING RULES

Rule-2: Decomposing part of a composition

If a new component, $C$, is required to be composed with an existing composition of $AB$, as depicted in Fig. 7.6a, to become as in Fig. 7.6b, only a heterogeneous type of composition connector is considered. As a consequence of composing $C$ into $AB$ as in Fig. 7.6b, a new lower-level hierarchy together with its composition points are created. In this case, the design cannot be further refactored because both the connectors are heterogeneous connectors, hence rule AR-1 cannot be applied here. By adding a lower-level hierarchy, this strategy justifies and motivates the incremental composition tasks discussed in Chapter 4.

This rule can be represented as:

$$Arch_{ori} = CC_1(C_A, C_B)$$

$$Arch_{target1} = CC_1(CC_2(C_C, C_A), C_B) \quad (7.3)$$

$$Arch_{target2} = CC_1(C_A, (CC_2(C_B, C_C))) \quad (7.4)$$

where $CC_{1..n}$ = heterogeneous composition connector, $C_{A..C}$ = component or composition and $Arch$ = architecture. In this case, $CC_1$ and $CC_2$ must be instantiated from a different type of connector.

The above case can also be represented as depicted in Fig. 7.7. The original
architecture of Fig. 7.6b can be of any ordering of combination of these two cases
(1) sequential i.e. SEQ or PIPE or (2) selection that is using SEL connector. Let
us observe the possibilities of adding a lower hierarchy for both the cases. In the
first case, let $CC_1$ be a SEQ, and $CC_2$ be a SEL. The original architecture can be
represented as in Fig. 7.7a, where the execution flows from component A, then
B. This is the same representation in Fig. 7.6a. Whenever a new C component
is composed using a new connector, $CC_2$ (see Fig. 7.6b), the control execution
can also be modelled as shown in Fig. 7.7b. Observe that the behaviour of the
original execution in Fig. 7.7a is indeed preserved.

![Fig. 7.7: Representing execution flow for Sequential and Selection Cases: Case 1](image)

The second case is to let $CC_1$ be a SEL instead, and $CC_2$ be a SEQ. We are
still referring to the original architecture in Fig. 7.6a, where the execution flows
from component A, or B. The same representation can be modelled as in Fig.
7.8a. Whenever, a new component, C is composed using a new connector, $CC_2$
(see Fig. 7.6b), the control execution can also be modelled as shown in Fig.
7.8b. This time, note that the behaviour of the original execution in Fig. 7.8a
is not preserved. There is no other means of preserving the original execution if
component C is to be added and sequenced before A. The only valid composition
point will be sequencing after \( A \). Hence, the refactoring of this structure is not permitted.

![Fig. 7.8: Representing execution flow for Sequential and Selection Cases: Case 2](image)

Now, let us consider another case which refactors from original architecture Fig. 7.6a to Fig. 7.6c. The same cases applicable here, which can be of any ordering of combination of cases (1) sequential i.e. SEQ or PIPE or (2) selection that is using SEL connector. The following discussion provides the possibility of adding a lower hierarchy for both cases. In the first case, let \( CC_1 \) be a SEQ, and \( CC_2 \) be a SEL. The original architecture can also be represented using a UML activity diagram as in Fig. 7.9a, where the execution flows from component \( A \), then \( B \). This is the same representation in Fig. 7.6a.

If a new component, \( C \) is composed using a new connector, \( CC_2 \), in this case a SEL (see Fig. 7.6c), the control execution is exemplified in Fig. 7.9b. Observe that the behaviour of the original execution in Fig. 7.9a is not completely preserved. Sequential execution from \( A \) can only be proceeded if the selection criteria is satisfied. In order to allow such a preservation of behaviour, the condition must always be satisfied, which means that component \( C \) may never be executed. This violates the aim of the intended design.

The other case for refactoring of the original architecture Fig. 7.6a to Fig. 7.6c
would be the case where $CC_1$ is a SEL, and $CC_2$ is a SEQ. An activity A or B components based on the selection criterion. The same representation is presented in Fig. 7.6a. It is worth noting that the connectors in XMAN exclusively deal with communication and coordination amongst components, instead of within computations in a single component [VL10].

If a new component, C is composed using a new connector, $CC_2$, in this case a SEQ (see Fig. 7.6c), the control execution is exemplified in Fig. 7.10b. Observe that the behaviour of the original execution in Fig. 7.10a is accurately captured and preserved.

The above cases show that whilst making design decisions to refactor, a designer should be concerned on the central aim in refactoring that is to maintain behaviour preservation. The following discussion provides example for adding a lower level hierarchy.

Example. Let us look at a design from the VSS [LSB98] example. In this design, the original architecture consists of a composition of Return Tape (RT) and Return Tape (RTN) components using a SEL. See Fig. 7.11a. Consider if a
new behaviour is relevant to one of the connected components. What is a possible way to handle this?

By allowing the addition of a lower-level hierarchy, the target composition could be more descriptive and meaningful. In this case, the designer must ensure that the original behaviours are still preserved. As such, the selection criterion must still be made available externally. For instance, originally, the selection of the composition is between RT and RTN. Even after the refactoring task of adding a lower-level hierarchy (see Fig. 7.11b), the designer must ensure that the selection
criterion is still available. In this example, even after the refactoring, the selection is still made between the two options.

The issue here is whether a new component (with additional behaviours) needs to be replaced for the composition. Once a composition is made, the composition is accordingly fixed. In order to replace part of a composition, another rule is required. By adhering to this particular rule, any replacement made to an existing composition can guarantee the behaviour preservation feature, instead of making an arbitrary replacement.

Summary → 2 Adding a Lower-level Hierarchy

AR-2 Decomposing an existing composition will result in adding a new lower-level hierarchy.

7.3.1.3 Adding an upper-level hierarchy

Assume that later in the incremental step, a new component, C, is required to be composed with A using another heterogeneous connector (that is from a different connector type), and this will result in creating a new upper-level hierarchy. In this case, there are two possibilities, either composition happens at the front (see Fig. 7.12b) or at the end (see Fig. 7.12c) of the existing architecture.

![Fig. 7.12: Adding an upper-level hierarchy](image)

This rule can be represented as:
7.3. ARCHITECTURE REFACTORING RULES

\[ Arch_{ori} = CC_1(C_A, C_B) \]
\[ Arch_{target1} = CC_2(C_C, (CC_1(C_A, C_B))) \] (7.5)
\[ Arch_{target2} = CC_2((CC_1(C_A, C_B)), C_C) \] (7.6)

where \( CC_{1..n} = \) heterogeneous composition connector, \( C_{A..C} = \) component or composition, \( Arch = \) architecture.

This rule is used to motivate refactoring tasks that result in creating an additional hierarchy level. It is relevant whenever heterogeneous connectors are involved, otherwise, the available open composition points can be applied if we are dealing with existing homogeneous connectors. This rule is in line with the normal composition defined in the X-MAN model.

**Summary → 3 Adding an Upper-level Hierarchy**

AR-3 Adding an upper-level hierarchy can be achieved by adding a heterogeneous connector to an existing composition.

### 7.3.2 Horizontal Refactoring

The aim of horizontal refactoring is to improve architectural design. This includes removing unnecessary architectural elements due to redundancies such as multiple guards or any duplicate behaviour. The only refactoring rule specified under the horizontal refactoring category is to replace the parts of an architecture based on a pre-defined set of behavioural patterns.

#### 7.3.2.1 Replacing parts of an architecture

In order to simplify an architectural design, a designer identifies any relevant behavioural pattern that can be used to replace parts of an architecture. This pattern can be replaced with a better design in the context of making the design more descriptive and reducing guards. Currently, a pattern consisting of PIPEs
and guards has been identified. As a result of replacing this pattern with a target architecture, the resulting architecture may also increase the hierarchy levels. With the existence of guards, each time a composition is made, the former guards must also be considered to be propagated.

**Rule-3: PIPE with Guards Pattern**

If a design contains a PIPE with guards that are checking the same constraints with different expected results i.e. *mutual exclusive*\(^4\) (see Fig. 7.13a and Fig. 7.13b, the architecture can be refactored into a SEL connector.

![PIPE with Guards Pattern](image)

(a) PIPE with Guards (b) Alternative representation

Fig. 7.13: Refactor PIPE with Guards

In this generic case, a SEL can be adopted whenever the same constraints need to be evaluated; the resulting value is *mutual exclusive* and will branch to a different component when a different value is received. It is worth noting that these refactorings are relevant whilst accommodating the proposed incremental approach. When dealing with each requirement, in each increment, the designer models the extracted keywords from NLR into architectural constructs.

During these increments, the resulting design of the system architecture is a result of the composition of these partial architectures. Without the incremental approach, a designer might not encounter the same scenarios. In an approach

---

\(^4\) Only one of the constraints will be true, hence, only a single component will be selected and executed at one particular time.
where a set of requirements are analysed as a whole, the intuition and judgment of the designer will be applied during the abstraction process. The derived design from such a process is typically based on a series of refinement process.

The following examples demonstrate the refactoring for replacing parts of an architecture. These examples have manifested the use of this refactoring during the execution of the case studies provided in Appendices C.

**Example 1.** In Fig. 7.14a, all the guards are checking the same approval status from the authentication process. If the approval status returns true, the selected transaction based on the corresponding user choice will be executed.\(^5\)

![Fig. 7.14: Pipe with Guards Pattern.](image)

(a) Guards with the same constraints

(b) Refactored design

The earlier design can be refactored by treating all the guards that check the same constraints and have the same result (approve status is true) as a \(\text{SEL} \). When these guards are changed, in order to preserve the behaviour of the previous composition, a single guard (instead of many guards in the previous architecture) that checks the constraint on top of the \(\text{SEL} \) is added. Hence, the resulting composition can be shown in Fig. 7.14b. By performing this step, an evaluation of the constraint (which is encapsulated by the guards) can be performed once. In contrast, in the original architecture, the evaluation is done for each branch.

\(^5\)For simplicity, the choices are not labelled here.
Thus, the redundant guards can be combined and the number of evaluations can be reduced.

However, the rule is only applicable to constraints that allow options to be selected between the guarded branches. If both of the branches need to be executed, then this rule is irrelevant. An example of such a case can be shown in Fig. 7.15 (drawn from the Video Store System (VSS) example). In this case, both of the branches must be executed if the constraints are satisfied.

Looking back at the Example 1 (see Fig. 7.14b), as a solution, the guards that evaluate the same constraints can be aggregated and encapsulated into a single guard. This guard needs to be put at a valid composition position. In this case, the guard is brought up front so that the constraint can be checked before the execution of the SEL.

Example 2. Let us assume that the guards are checking the same constraint and some of the branches may handle different results. See Fig. 7.16a. In this case, if the returned approval status is false (which means the authorisation has failed) then the reject card computation will be invoked. This computation is provided by the Reject Card (RC) component. The rest of the guard branches have the same value, i.e. true. In such a case, the design can be transformed into two different SELs that encapsulate two different constraints.

In this example, a SEL is added to replace all the guards that are checking the same constraints and having the same result, that is the valid approval status. See Fig. 7.16b. However, the branching for the RC component must be designed in a separate SEL. As a result, another SEL is added at a higher hierarchy level.
7.3. ARCHITECTURE REFACTORING RULES

![Diagram of Original Architecture](image1)

![Diagram of Target Architecture](image2)

(a) Original architecture  
(b) Target architecture

Fig. 7.16: Before and after refactoring

so that the same behaviour can be represented.

It is important to note that as this approach is incremental, hence it is highly possible that the architecture design will contain many guards as an effect of the incremental process. By applying this rule, these unnecessary guards can be reduced. As a result, the refactored architecture will be more structured and meaningful, and hence, can help to reduce the complexity of the structure. Although the resulting architecture (by the means of refactoring the guards into a SEL connector) has an additional hierarchy level instead of reducing hierarchy levels, the evaluation of the guards is now reduced. In this case, the evaluation of the approval status can be performed once and then proceeds to the execution of the selected transaction. Such a design is closer to the realistic behaviour stated in the requirement when they are analysed as a whole. In contrast, if one particular requirement is being dealt with individually, this issue might not be a concern.

Note that the horizontal refactoring may have an impact on vertical refactoring tasks. When this rule is applied, additional hierarchy levels are created. This step is necessary to build descriptive architectural design, remove redundancies
and reduce unnecessary effort.

**Example 3.** Fig. 7.17a shows that the architecture comprises a PIPE and two guards that are checking the same constraints. In this example, if the file validation returns false, which means that the validation fails, the Display Error (DE) component’s computation will be invoked. Otherwise, if the validation returns true, which means the validation is approved, only then the Format File (FF) component’s computation will be invoked. If the AR-4 rule is applied here, both of these guards can be simplified and replaced by a SEL connector as shown in Fig. 7.17b.

![Diagram](image)

(a) Original architecture  
(b) Target architecture

Fig. 7.17: Refactor guards — WordCount

**Summary → 4 PIPE with Guards Pattern**

**AR-4** PIPE with mutual exclusive guards can be simplified into a PIPE with SELs.

### 7.4 Issues and Discussion

The following are a few anticipated challenges and issues pertaining to the architecture refactoring process.
7.4.1 Correctness criterion

In order to guarantee behaviour preservation, a correctness criterion has to be decided. The correctness criterion evaluates both the original and the refactored architecture to ensure that the same behaviours are maintained. So far, the adopted criterion is based on analysis and justification. In order to ensure behaviour preservation, work such as [MQR95] adapted faithful interpretation, [TB01] adopted database schema evolutions, whilst [Opd92] provided invariants and enabling conditions. However, all of these works applied the refactorings to object-oriented design and implementation.

7.4.2 Measuring design complexity

Currently, no formal software metrics are used to measure the complexity of the design. In general, the number of connectors are relative to the complexity level of the architecture. Thus, our aim is to reduce the number of connectors and guard adapters whenever possible, while maintaining the behaviour preservation.

7.4.3 Handling components' interfaces

A component's interface is represented as an abstract element that exposes the services provided by the component. Typically, the interface is specified by its method signature, consisting of the name of the service and its required input and output (including type and quantity). By exposing these services, any composition connector will be able to coordinate control between components. However, when refactoring happens, careful consideration has to be put into action, as the main aim is to maintain behaviour preservation in each increment. Each time an architecture is refactored, the necessary updates to its corresponding interfaces must be taken care of. Without this consideration, the refactored part of the
architecture might not be accessible.

7.5 Summary

Architecture refactoring supports the changes to system architecture structure while maintaining the preserved behaviours. In order to achieve behaviour preservation whilst satisfying the required behaviours, the set of architecture refactoring rules could assist the system designer in making design decisions. In this chapter, the architecture refactoring rules together with their justifications and examples are discussed and presented. In Chapter 8, the formulation of the entire approach to constructing component-based systems from NLR will be presented.
Chapter 8

Defining the Approach to Constructing Component-based Systems Directly from Requirements

The primary aim of this chapter is to set the scene for the entire approach in regards to the construction of component-based systems from requirements. This approach consists of a five-step process involving extracting elements of component-based systems, the mapping of the extracted elements into the X-MAN constructs, creating a partial architecture, creating an incremented system architecture and finalising the architecture. The secondary aim of this chapter is to demonstrate the feasibility of the approach. In an attempt to perform such a demonstration, a simple yet complete ATM example will be adopted. In addition, a larger set of requirements (Trading System (COCOME) [RRMP08]) will be applied in the subsequent chapter so as to evaluate the validity of this approach.
8.1 Introduction and Motivation

In this approach, only a single requirement\(^1\) is being dealt with at a single point of time. The keyword extraction process is carried out for a single requirement at a time. This process is possible because of the encapsulation property in the X-MAN component model. Moreover, it is an advantageous feature because analysing one requirement is more manageable than the usual practice of analysing all requirements together. In addition, dealing with each requirement can also lead to scalability (in terms of dealing with any number of requirements), and always a finitely terminating process.

With such an incremental approach, each requirement is handled in a systematic way, hence reducing the abstraction effort made during the analysis and design stages. These abstractions are mostly subjected to the expertise, intuition and judgment of the designer’s efforts. During these processes, it is possible for arbitrary decisions to be made.

In order to achieve the above-mentioned aim, and also to demonstrate the complete cycle of the approach, it is therefore plausible to apply a simple, yet complete example that is the ATM System. This strategy allows us to focus on the processes involved in each step of the approach and at the same time understanding the entire approach.

As the basis of an approach to constructing component-based systems directly from requirements, a component model is adopted. Based on the analysis of the existing mapping approaches as laid out in Chapter 2, to the best of our knowledge, there is no work that deals with one requirement at a time apart from the Behavior Engineering (BE) approach, and at the same time allows the construction of a system incrementally. In this work, a component model is

\(^1\)It is worth noting here that a single requirement is not necessarily one statement.
employed as the basis of the approach to constructing systems directly from requirements. This approach differs from the BE approach. The BE approach builds the behaviour tree of the desired system, extracts a system structure from the behaviour tree and a series of refinement diagrams. The DBT diagram aids in the derivation of a component diagram. The construction of the component diagram is therefore in a single step and not performed incrementally. It then generates codes for the system.

In addition, the BE approach does not have the notion of pre-existing components, but instead generates all the codes for every system from scratch. In principle, a component-based approach is better not only because it enables the re-use of pre-existing components, but also because it facilitates the construction of the system architecture incrementally. The BE approach builds the behaviour tree for the system incrementally, but not the system itself.

In this incremental approach, by applying the pre-defined sets of heuristics and the design decision rules, this strategy aids the system analysts or designers in making design decisions throughout the construction process. In contrast to merely relying on their expertise or experiences that may lead to constructing systems arbitrarily, these heuristics and rules can contribute to a systematic semi-formal process.

8.2 Defining the Approach

As introduced in Chapter 1, this work attempts to map requirements into component-based systems using incremental composition. The conceptual model (see Fig. 8.1) depicts the overall context of the approach. With requirements as the input, being extracted and mapped to component-based systems and incrementally be built to form a system architecture.
Based on the semantics of the X-MAN component model, the design decisions and heuristics for extracting elements of component-based systems, the incremental composition and the architecture refactoring features, a system designer (or an analyst) adopts this approach with the intention of constructing component-based systems directly from NLR. Fig. 8.2(a) illustrates the process model involved. As requirements may contain problems such as incompleteness, ambiguities and inconsistencies, the designer shall consult with the system users to clarify these issues. It is beyond the context of this work to provide any solution to such issues.
8.2. DEFINING THE APPROACH

At every incremental step, each requirement is executed based on the steps as illustrated in Fig.8.2(b). Accordingly, the subsequent discussion will be presented and organised according to these steps. The following subsections discuss each of the processes involved.

8.2.1 Defining Algorithms for the Incremental Approach

In the incremental approach, requirements are dealt with one at a time. Based on Algorithm-1, for each requirement, firstly, element extraction process is performed and then the result of the extraction process will be mapped into modelling elements as defined in the XMAN semantics. The detailed algorithm for the extraction process and the mapping process are shown in Algorithm-2 and Algorithm-3 respectively.

Algorithm 1: The Incremental Approach to Constructing Component-based Systems Algorithm

\begin{verbatim}
Require: A set of requirement statements. Requirements are dealt with one at a time.
Understanding of the XMAN semantics, the extraction and mapping heuristics are required.
Ensure: A system diagram using XMAN semantics.
while requirement n is in sequential order do
  if requirement n is not the last requirement then
    Perform element extraction process. See Algorithm-2.
    Map the extracted elements to the component-based elements. See Algorithm-3.
    Increment requirement n + 1
  end if
end while
\end{verbatim}

It is worth noting that the extraction process i.e. Algorithm-2, is grounded by the defined set of terms i.e. computation, control and data based on the XMAN component model semantics. The pre-conditions for performing the element extraction process are (1) availability of the requirement (R.Q) to be extracted and (2) knowledge of the XMAN component model and understanding of the element extraction heuristics. These heuristics are provided as a guideline to be applied during the extraction process.
**Algorithm 2: The Element Extraction Algorithm**

**Require:** A requirement (RQ) to extract the candidates of architectural elements.
Understanding of the XMAN semantics and the extraction heuristics are required.

**Ensure:** Extracted computations, control and (or) data based on the XMAN semantics.

Read each RQ element

- **if** (RQ element is a verb) **then**
  - **if** the verb is (data transformation || state || event) **then**
    - Extract computations using Heuristic HR-1A.
  **end if**

- **else if** (RQ element is a noun) **then**
  - **if** (if the noun is an action noun) **then**
    - Extract computations using Heuristic HR-1A.
  **end if**

- **else if** (RQ element is (descriptive phrases || predicate)) **then**
  - Extract computations using Heuristic HR-1A.
  **end if**

- **if** (RQ element is (user interaction || implied from hardware services)) **then**
  - Extract computations using Heuristic HR-1B.
  **end if**

- **if** (RQ element is implied from conceptual components) **then**
  - Extract computations using Heuristic HR-2.
  **end if**

- **if** (RQ element is (preposition || conjunction || pre-defined control terms || implied from preposition)) **then**
  - Extract control using Heuristic HR-3.
  **end if**

- **if** (RQ element is (noun AND relevant)) **then**
  - Extract data using Heuristic HR-4.
  **end if**

Get the extracted computations, control and data.
8.2. DEFINING THE APPROACH

For each of the requirement, the main aim is to extract relevant keywords that may denote (1) computations, (2) control and (3) data. These three elements are fundamental elements in XMAN semantics. Functional requirements generally state the desired behaviours of the system to be developed. The behaviours which we are interested in are data transformations. Nonetheless, these may be explicitly stated, hence, we need to identify keywords that denote computations from the requirements. These computation keywords can be identified from many sources such as data transformation, state, event, action noun, in descriptive description or predicate expressions. For such purpose, users may use Heuristic HR-1A.

Even if a computation is found in the former case, users are still required to look for any potential computation candidate in the remaining parts of the requirement statement. The same case also applies for control and data extraction. In the cases where keywords that denote computations may not be explicitly stated, computations can be extracted based on (1) user interaction with the intended system or (2) implied from hardware services. These cases have been addressed in Heuristic HR-1B. Lastly, for identifying keywords that denote computations, users may also encounter conceptual components, in which components are explicitly being stated without explicit data transformation. To handle such a case, users can refer to Heuristic HR-2.

In order to identify a control, which may be originated from a pre-defined set of selected prepositions, conjunctions, control terms or implied from prepositions in the natural language requirements, users may adopt Heuristic HR-3. The final part of the extraction process is the identification of keywords that denote data. Heuristic HR-4 is created to handle the data issue. Once the extraction process is completed for a particular requirement, the next step is to perform the mapping process.
The following process involves the mapping of the extracted elements to component-based system elements. The detailed algorithm for the mapping process is shown in Algorithm-3. The pre-conditions of the mapping process are (1) result of the extraction process, (2) the understanding of the XMAN semantics and (3) knowledge of the mapping heuristics to be applied during the mapping process. The first design decision to consider is whether the extracted computation elements should be encapsulated in a single or separate components. Highly encapsulated components could increase the probability of matching components in repository.

As this approach is applied in the deployment phase of the component-based system life cycle, all the mapped components are assumed to already be existed in a component repository. If in any case that there is no matching component, it is assumed that the component must be priorly designed and created. The process of creating a new atomic component in component-based development is considered as design phase, which significantly differs from the deployment phase.

The next process is to map the extracted control keywords to its corresponding composition connectors i.e. SEQ, PIPE or SEL or adapters i.e. Guard or LOOP. These design decisions are guidelines for the users to adopt whilst making decisions on which component-based element to be mapped for the extracted control. An extracted control keyword from a particular requirement can only be mapped to one of the defined control element i.e. SEQ, PIPE or SEL or adapters i.e. Guard or LOOP.

The third main step is to map the extracted data keywords to data semantics defined in XMAN. In XMAN, relevant and useful data are data that are used in computations. Hence, the user should identify the relevant data based on the selected computations to be performed.
8.2. DEFINING THE APPROACH

As exceptions, cases where (1) no computation is extracted from the extraction process; or (2) no indication of control, are also being considered. The former case covers situations in which no explicit computation could be extracted from the corresponding requirement statement. As a solution, an inference should be made so as to derive computations from each requirement. In this approach, only functional requirements are applicable. Hence, although there is no explicit computation from the requirement, it is anticipated that each functional requirement should contain the relevant computation(s). The latter case occurs when there is no explicit control terms to be extracted from the requirement. Here, it is assumed that the order the requirement is written represents the ordering execution of the extracted computations.

The subsequent task during mapping the extracted component-based elements is to model them using XMAN model. The first design decision to be made is where should the composition be. Whenever we anticipate that the existing composition connector is highly reusable, we adopt a new composition connector. A connector can be considered as potentially reusable whenever based on the designer's judgment, there is a possibility that in the later steps, the same control structure can be reused. For instance, if we design a PIPE with a specific type of input data and is expected to have a specific number of ordering execution, we can store this composition connector's configuration for later purposes. Using this strategy, we allow the existing composition to be used in further compositions. On the contrary, if we do not anticipate that the existing connector might be reused, we can directly use the existing composition connector as long as the expected behaviour is satisfied. The final outcome from the mapping process is the final system diagram consisting of all the mapped components and composition connectors.
Algorithm 3: The Mapping of the Component-based Elements to the XMAN Elements Algorithm

Require: The extracted computations, control and (or) data from Algorithm-2.
Understanding of the XMAN semantics and the mapping heuristics are required.
Ensure: A final system diagram based on the XMAN semantics.

if (computation keywords are extracted) then
    Assign computations to its relevant component.
    Use Heuristic HMR-4B.
    if (more than one computations are related) then
        Encapsulate them in a single component.
    else if (computations are independent) then
        Assign each computation in separate components.
    end if
end if

if (control keywords are extracted) then
    if (the control keywords are based on ordering execution) then
        Apply Heuristic HMR-1.
        if (data is required) then
            Apply Heuristic HMR-2.
            Apply a PIPE.
        else if (data is not required) then
            Apply a SEQUENCER (SEQ).
        end if
    end if
else if (control is based on branching execution) then
    Apply a SELECTOR (SEL).
else if (control is constrained on a single component) then
    Apply Heuristic HMR-3.
    Apply a GUARD.
end if

if (data keywords are extracted) then
    Map relevant data to be used.
end if

if (no explicit computation keyword is provided) then
    Use computation inferences.
    Apply Heuristic HMR-5.
end if

if (no explicit indication of control keyword is provided) then
    Assume the order the requirement is written.
    Apply Heuristic HMR-7.
end if

Compose the mapped elements using XMAN modelling semantics.
if (composition connector is potentially reusable) then
    Apply Heuristic HMR-4A.
    Use a new composition connector.
else if (composition connector is independent) && (the behaviour permits) then
    Use an existing composition connector.
end if
8.2. DEFINING THE APPROACH

8.2.2 Extracting Elements of Component-based Systems from Requirements

In the light of the entire approach, first, keywords of the component-based systems from each requirement statement are analysed, identified and extracted, as detailed in Chapter 5. In the extraction process, an analyst (or a designer) examines each of the words in a requirement statement and identifies the computations, control and data. From a requirement, by using the Extractor tool, the analyst highlights the extracted verbs, nouns and control terms. The control terms here are defined based on a selected list of prepositions, conjunctions, and terms that may denote control. The extraction process is done with the help of the POS tagger and the pre-defined set of heuristics.

First, the analyst selects the computations based on the semantics of data transformation (not just any operations or verbs). The analyst then picks keywords that denote control, and finally, he decides on any relevant data pertaining to constraint data, selection data, initialisation values or values to be passed to any external computation. These values are not handled by the identified computations but rather by providing the values to be used in other computations. The output of this step is extracted computations, control and data.

As a summary, for each requirement, the keyword extraction process comprises the following steps:

1. Execute the Extractor tool to perform parts-of-speech (POS) tagging on the requirement and also to extract selective types of verbs, nouns, prepositions and conjunctions. A POS tagger is able to parse a piece of text and extract words corresponding to the POS specified by rules defined by the user.
2. Analyse the results of the POS tagger category by category; for each category, keywords are further analysed and filtered according to pre-defined heuristics.

3. Identify computations, control and data.

8.2.2.1 The Extractor Tool

The Extractor tool has been developed in order to assist the analyst in identifying and extracting the elements of component-based systems. The main advantage of the tool is that it abates the amount of text to be digested by an analyst while dealing with a single requirement. It is not claimed that this tool can be used for automatic extraction, but it is merely a promising tool to assist analysts in identifying component-based elements from NLR based on a specific component model, i.e. the X-MAN model.

Fig. 8.3 depicts the flow of the Extractor tool. Initially, a requirement will be used as an input to the tool. Each word will be syntactically tagged using a pre-defined selected POS tagger set. The X-MAN component model defines what are components, connectors and the composition mechanism. Based on these notions, the elements to be identified from text are derived. The main keyword elements are computations, control and data. In order to syntactically extract these elements, the Brown Corpus POS tagger\(^2\) is adapted and adopted.

---

\(^2\)http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html. A brief introduction on
8.2. DEFINING THE APPROACH

Fig. 8.4 shows one of the screen shots of the Extractor tool. Initially, a requirement will be an input to the tool. Each word will be syntactically tagged, using a pre-defined selected POS tagger set. The tool includes the built-in heuristics which allow the filtering of irrelevant words. For instance, articles (e.g. ‘the’, ‘a’, ‘an’) will not be extracted. Moreover, a user can highlight words according to verb, noun or control features (see Fig. 8.4). The tool thus helps by suggesting keywords that may denote control, computation or data based on the POS tagger and some pre-defined rules. Nonetheless, the analyst needs to manually filter and finalise all the selected keywords.

A dictionary to document all the extracted elements is highly recommended, in order to reduce risks of redundancies or inconsistencies in the extracted elements. To support this, the Extractor tool provides a simple means of specification for the extracted elements, and this specification can be used as a reference for the succeeding processes.

8.2.3 Mapping of Extracted Elements to X-MAN Elements

Having all the extracted elements that are the computations, control and data, the subsequent task is to map these elements into the X-MAN elements. The entire mapping process has been discussed in Chapter 6. Based on the guidance provided in that chapter, each of the extracted keywords that denote computations, control and data are assigned to their respective X-MAN elements. If a

the tagger has been included in the Appendix A.
keyword that denotes control is extracted, for instance, it is mapped into a corresponding X-MAN element, i.e. a composition connector or an adapter. This process is manually performed by the analyst (or the designer). In this step, first, the identified computations are assigned to their corresponding components. A component provides services which are the computations that it contains. During this task, the use of descriptive names to specify these elements will be useful. In order to avoid redundancies, each component together with its provided services and identified data must be specified and stored in a dictionary.

Whenever more than a single computation is extracted, these computations are assigned to their logical providing component. The following step is to identify the control or flow of execution between these components. These controls are mapped to either composition connectors or adapters, and finally the relevant data are mapped to data elements of component-based systems.

Once all the components are identified in each increment, they are selected from an existing repository. This repository contains pre-existing components, generic composition connectors and adapters. The selected components, connectors and adapters will be instantiated and customised accordingly. In this approach, it is assumed that these components will be selected from these pre-existing components in the repository. However, when a suitable component cannot be found, this component needs to be built first and deposited into the repository. The development of such a component is supported using a completely separate component-life cycle development process.

An exceptional case is when there is no indication of any computation from a particular requirement. The implication of not having any computation as a result of the elements extraction is that the architecture cannot be derived from the requirement. This may be due to a few possible reasons. Firstly, statements that contain data elements and their relationships only. For instance, "A course is
taught by one or more lecturers.’’ Such a statement is rather useful in database modelling, but since this approach is a behavioural approach, only functional requirements are considered. Secondly, statements that contain relationships information. For example, ‘‘A student can be an undergraduate or a graduate.’’ In object-oriented modelling, such information may be regarded as inheritance properties, however such information is not a concern in this approach.

Another exceptional case is when there is no indication of any control from the requirement statement. The impact of not having any connector as a result of the elements extraction is that the components could not be composed, even if computations have been identified from the requirements. A possible reason that may contribute to this issue is when a statement only describes a single computation. Hence, only a single component will be identified without any control. A foreseen issue here is finding the valid composition point when the mapped element is to be composed with the incremented system architecture. This is when an assumption is applied that each statement is written in an order even without any explicit indicator. For instance, ‘‘A user enters card. A user enters PIN.’’ Here, although there is no clear indication of any kind of ordering, it is assumed that the prior statement is executed first.

Whenever a constraint data is extracted and even without an explicit hint for an adapter, in most cases, a guard adapter is required. Otherwise, there will be no component-based element that will handle the evaluation of the constraint data. This issue has been address during the mapping of the extracted elements of component-based process in Chapter 6.

8.2.4 Creating partial architectures

Subsequently, in the third step, the mapped elements are used to model a partial architecture based on the X-MAN compositional semantics. This process requires
in-depth understanding of the X-MAN model. A partial architecture represents the behaviours stated in a requirement. The results of element extraction and mapping to the X-MAN constructs steps should be sufficient to enable a partial architecture for the requirement to be constructed.

Ideally, in each increment, only a single requirement is being dealt with. The created partial architecture will be incrementally composed with the previous incremented architecture in the following step. In this step, modelling of the component-based elements is devised based on the X-MAN component model's notation and its semantics as presented in Chapter 3.

In brief, components and composition connectors are modelled. These connectors are used to compose components in order to build an architecture. In this thesis, this architecture is referred to as a partial architecture. This partial architecture is built for a particular requirement based on the mapped X-MAN elements.

8.2.5 Composing partial architectures with the existing system architecture

In the following step, the constructed partial architecture which is derived from the previous step is composed to the existing system architecture. This system architecture will be built-up gradually by composing partial architectures with the incremented system architecture. During this stage, the rules for composing the partial architecture with the system architecture are based on incremental composition and architecture refactoring rules. By adhering to these rules, the desired characteristics (i.e. encapsulation and compositonality) of the derived architecture may be achieved. As such, the resulting architecture can be further

\footnote{For the initial increment, the partial architecture is treated as the initial system architecture.}
composed with other composition units, which adhere to the X-MAN semantics. The significance of the refactoring rules during this step is to resolve any composition issues, in addition to the incremental composition mechanism.

In order to perform this step, the incremental composition concept comes into action. This concept has been discussed in Chapter 4. This chapter demonstrates how the incremental composition supports and facilitates the composition of components and compositions into an existing architecture.

Nonetheless, this step may not be possible if the partial architecture of the current requirement cannot be related to the current system architecture. This can easily happen, as the requirements document is unstructured. When this happens, the incremental composition for this requirement needs to be deferred until it becomes clear where to compose such an architecture. If it never becomes possible, then as with all requirements, there may be problems with the requirements themselves, and the system user needs to be consulted in order to resolve any ambiguities, inconsistencies or incompleteness of the requirements.

In each increment, there may be many possible composition points (denoted by ‘…’ in an open composition connector) associated with many open composition connectors. A valid composition point must be chosen in order for the composition to achieve the behaviours that are stated in the requirement. In particular, once the composition connector is chosen, a decision has to be made as to whether a new component can be added at any composition point in the connector, or whether it must be added at the front or at the end of the other existing components composed by the existing connector. This decision, of course, depends on the expected behaviours as stated in the requirement.

These steps (first to fourth) are performed in each increment, that is, for each requirement until the final requirement is dealt with. Once the partial architecture which derived from the final requirement is composed with the system
architecture, the final system architecture is considered as accomplished. In addition, this is when the architecture refactoring tasks can be applied. The main aim of the refactoring rules during the final step is to simplify the derived architecture, whilst maintaining the original behaviours. These rules have been discussed in Chapter 7.

8.2.5.1 The Exogenous Composition Framework Tool

The Exogenous Composition Framework (ECF) tool has been developed to assist the systems designer in modelling and composing the partial architectures using incremental composition. Rules that can automatically be checked by the tool have already been embedded; hence, using this tool can help to reduce other unnecessary constraints during composition, and the designer's effort could be strictly focused on issues of incremental composition.

Fig. 8.5(a) and (b) show the screen captures of the ECF tool. The value of such a tool is clearly that it enables the designer to model the architecture, construct systems and then execute them. This is only possible when all the required components and generic composition connectors already existed in the repository. This way the composition of the architecture is considered executable. This tool also enables the designer to validate a final system architecture with respect to the system's requirements. By validating the system against a set of test cases, this strategy contributes as one of the means to provide evidence that the system satisfies its requirements. The usage of the tool, in general also experimentally validates the approach of the system construction directly from requirements.

The ECF tool supports component composition (both in design and deployment phase) according to the X-MAN component model's semantic. However, as this approach only works in the deployment phase, the discussion is constrained
8.2. DEFINING THE APPROACH

(a) R1 (b) \{R1, R2\}

Fig. 8.5: The Exogenous Composition Framework Tool.

to this phase only. The result of this composition can be used to generate Java codes accordingly.

The tool provides two views: (1) the design view, which is to support modelling of an architecture; (2) the code view, which is to translate the model produced in the design view into Java codes in order to be executed using the X-MAN model’s Application Programming Interface (API). As shown in Fig. 8.5(a), CardReader (CR) and PinReader (PR) components are selected from a repository of pre-existing components, and the instances of each component are created. Then, a SEQ connector is selected from a repository of pre-existing composition connectors and an instance of the connector is created. The following step is to create the connections between all of the modelling elements in order to create a composition. A hollow circle that denotes an interface on top of the connector represents the open composition where more components are allowed to be composed with the existing composition. A solid interface, on the other hand, denotes the fact that the composition is fixed and thus no more components can be added to the composition.
8.2.6 Finalising the system architecture

When all the requirements have been mapped, an architecture for the entire system is produced. However, this architecture still has available composition points, and can therefore be refined, adapted or optimised. Components could be combined into larger composites; a set of connectors could be optimised to a single connector; connectors could be adapted by adaptors to add behaviour that is implicit in the requirements. The last step of the finalisation process is to remove any remaining available composition points, thus closing the whole (final) architecture. These architecture refactoring tasks have already been discussed in Chapter 7.

By finalising the composition points, all the dotted lines are removed and the lolly-pop interfaces are changed into solid ones. See Fig. 8.6(b). This indicates that any further incremental composition are restricted to be performed to the architecture.

In this step, the ECF tool supports the finalisation process by allowing a composition connector to be closed. As a result of this, the hollow lollipop interface will be changed into a solid lollipop interface. If an upper-level hierarchy is finalised, all the lower-level hierarchies will also be automatically finalised. This
8.2. *DEFINING THE APPROACH*

reflects the fact that no more composition is allowed, using the finalised composition connector.

In normal cases, the finalisation of composition connectors is only applicable during the final step because we could never be certain that the succeeding requirements may not change or alter any parts of the existing architecture. However, in an exceptional case where the designer is also a domain expert, he might be able to decide which composition point that can be finalised during each increment rather than waiting until the final step.

In addition, during this finalisation step, a loop adapter is added at the top level connector to suggest that the system is ready to be used by any system user. This can be shown in Fig. 8.7(b). Otherwise, the execution terminates once a cycle is completed or terminated. Note that a loop adapter is not a composition connector.

![Diagram](image)

**Fig. 8.7: Adding a Loop Adapter.**

**Example.** In the ATM example (see Fig. 8.7(c)), once the architectural design is completed, a loop adapter is added at the top level connector. This addition is to simulate a situation where once the system is executed, it is ready to be used by any system user.
8.3 A Complete Example: The ATM System

In order to demonstrate the entire approach, a complete ATM example will be used. The aim here is to lay out all the processes involved, hence leaving the details of each process to be referred to their respective chapters. For each requirement, all the presented steps will be applied and in the end the final system will be derived. This final system is executable, and can be regarded as a running system, provided all the required data are supplied.

8.3.1 Increment-1 - Requirement 1

R1: The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).

Step-1: Extracting Elements of Component-based Systems from Requirements

After executing the Extractor tool (see Fig. 8.8), for R1, user interactions with the system by inserting card and entering PIN are identified. These two interactions which imply computations, are performed sequentially. The system to be built must provide computations to read the inserted card information and the entered PIN. Note that the sequence must be preserved as performing the former prior to the latter interaction. For control, the ‘and’ conjunction is extracted, whilst for data, customer, card and PIN are required for the composed components (see Table 8.1 on the extraction result column). Nonetheless, there are also cases in which sequencing does not enforce any ordering execution.\footnote{See Chapter 5.} The detailed discussion on the extraction for R1 may be seen in Section 5.4.1.
8.3. A COMPLETE EXAMPLE: THE ATM SYSTEM

Fig. 8.8: Extraction of R1 using the Extractor Tool

Step-2: Mapping of the Extracted Elements to X-MAN Elements

For each computation, a candidate component, which is an existing component in a repository, will be assigned and selected. Based on the extraction information, components with read card and read PIN computations need to be provided.

Obviously, many computations can be grouped into a single component. Conversely, a single computation can also be separated into several components. For R1, it is decided that the computations are separated into two candidate components, namely the CardReader (CR) and the PinReader (PR) respectively. The main reason is that both the computations are device dependent, hence they can both be modelled as separate entities. These components are being looked up in a repository based on the signature matching of the components’ interfaces. In this case, the interface of CR component shows the component’s provided services.

Based on the identified control, the type of composition connector or adapter that is appropriate will be mapped: use a SEQ to sequence between computations; use a PIPE if data from one component is needed to be piped to the other components; use a SEL for branching. In addition, a guard or a loop can also be used for adapting a component or a composition.

By considering all the mapped elements for computations, control and data,
first, the computations are assigned with its corresponding components that provide the required services. Based on the preceding step, the system must provide computations to read the inserted card information and the entered PIN. Hence, these keywords are mapped to \texttt{readCard()} and \texttt{readPIN()} computations respectively. Here, the \texttt{readCard} computation is assigned to a \texttt{CardReader (CR)} component, whilst the \texttt{readPIN} computation is assigned to a \texttt{PinReader (PR)} component.

In addition, the ‘and’ conjunction is mapped to a sequential control flow between the two computations. Thus, the sequential execution is mapped to a \texttt{SEQ} composition connector. Finally, the extracted data keywords are mapped to the data elements in the X-MAN model. A summary of elements extraction together with their mapping to component-based elements for requirement R1 is shown in Table 8.1 in the mapping result column. These mapping design decisions are based on the heuristics presented in Chapter 6.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>Extracted elements</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>insert</td>
<td>DT [HR-1B]</td>
<td>\texttt{readCard()} computation \texttt{CardReader (CR)} component</td>
</tr>
<tr>
<td>enter</td>
<td>DT [HR-1B]</td>
<td>\texttt{readPIN()} computation \texttt{PINReader (PR)} component</td>
</tr>
<tr>
<td>Control and</td>
<td>Conjunction (ordering)</td>
<td>SEQ \texttt{[HM-2]} \texttt{SEQ(CR, PR)}</td>
</tr>
<tr>
<td>Data</td>
<td>customer, card, PIN</td>
<td>store customer, card, PIN data</td>
</tr>
</tbody>
</table>
Step-3: Creating partial architectures

The next step is to model the extracted elements according to the X-MAN component model’s notation. The result of this step is illustrated in Fig. 8.9. The composition shows that the control flow initiates from the top, invokes the readCard computation first, and then executes the readPIN computation.

Fig. 8.9: Partial Architecture for R1.

Step-4: Composing partial architecture with the existing system architecture

As this is the first partial architecture, the diagram is also treated as the initial system architecture diagram. This system architecture diagram will be incrementally constructed by composing more extracted elements identified from the subsequent requirements.

8.3.2 Increment-2 - Requirement 2

R2: A customer must be able to make a cash withdrawal from the linked account. Approval must be obtained from the bank before cash is dispensed.
Step-1: Extracting Elements of Component-based Systems from Requirements

Consider R2. From the result of the Extractor tool, the computation verbs (Table 5.2) approve, and dispense cash, and the computation noun (Table 5.3) cash withdrawal are identified as computations. Based on the HR-1A, the required elements are identified first, and then modelled accordingly. In R2, the approval, cash withdrawal and dispense cash computations are extracted. The subsequent step is to find hints for control. The main task here is to identify any hints for execution flow, either ordering, selection or repetition execution. First, the result of the authentication computation has to be passed to the other branch of composition before the cash withdrawal computation is allowed to be performed. In this statement, the ‘before’ preposition can be used as a hint to denote ordering (see Fig. 8.10).

![Fig. 8.10: Extraction of R2 using the Extractor Tool](image)

Step-2: Mapping of the Extracted Elements to X-MAN Elements

Then, these extracted computations are assigned to its corresponding meaningful component specifications i.e. Authentication (AUT) (for approve), Withdraw (WD) and DispenseCash (DC) components. The result of the approve computation has to be checked before the withdraw and dispense cash computations
are allowed to be performed. As a result, based on HM-3, a guard adaptor is required to evaluate the result of the authentication and invoke the computation only if the result is satisfied.

From the extracted elements in the previous step, the computations are assigned to their respective components that provide the required services, which are AUT (for approve), WD and DC components. The result of the approve computation has to be checked before the cash withdrawal and dispense cash computations are allowed to be performed. So a PIPE connector is used to compose AUT with a composite of WD and DC. The latter composite is the result of another pipe connector. A summary of the extracted elements and their corresponding mapped X-MAN elements is provided in Table 8.2.

Table 8.2: Summary of Steps 1 and 2 for R2.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>approval</td>
<td>Action noun</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>withdrawal</td>
<td>Action noun</td>
<td>Authentication (AUT) component</td>
</tr>
<tr>
<td>dispense</td>
<td>DT</td>
<td>DispenseCash (DC) component</td>
</tr>
<tr>
<td>Control</td>
<td>Preposition</td>
<td>PIPE (data dependency)</td>
</tr>
<tr>
<td>approval</td>
<td>constraint</td>
<td>PIPE(AUT,[approved])=PIPE(WD, DC)</td>
</tr>
<tr>
<td>Data</td>
<td>approval</td>
<td>Data (constraint) store approval status</td>
</tr>
</tbody>
</table>

For data extraction, when a constraint data is identified, accordingly a control for checking the constraint has to be added. In this case, as we extracted approval as constraint data, hence a new control for evaluating the constraint
is created. This constraint for handling the evaluation of the constraint is later mapped to a guard adapter.

**Step-3: Creating partial architectures**

The current system diagram for R2 can be shown in Fig. 8.11(a). Based on the analysis in the previous step, a guard adaptor for the composition of the WD and DC components is needed, because the control can only be allowed to reach the composition if the result of invoking the AUT component is positive.

**Step-4: Composing the partial architecture with the existing system architecture**

In normal composition, during each incremental step, a new connector must be used. As a consequence, the number of connectors and hierarchies will expand accordingly. This might lead to a complex design, with many connectors and hierarchies, and as a result, the architecture might be difficult to evolve and maintain.

![Diagram](Image)

(a) Partial architecture for R2  
(b) \{R1R2\}

**Fig. 8.11: Increment-2.**

The mapped elements for R2 can be composed to the existing partial system diagram with the feature supported by the incremental composition. In this
solution, a different type of connector, i.e. a PIPE, is identified, the extracted elements cannot in principle simply be composed with the existing composition connector i.e. a SEQ. Hence, the only way that the extracted elements in R2 can be composed is from the open interface of the partial system diagram. Fig. 8.11(b) exemplifies the new system architecture after the composition of the partial architecture for R1 and R2.

In this step, incremental composition is applied in order to compose the partial architecture for R2 with the current system architecture (in this case it is the partial architecture for R1). The results of the computations in R1 are needed for those in R2. Therefore, a PIPE is used to compose the two partial architectures, instead of a SEQ. The top-level PIPE in the partial architecture for R2 provides a suitable composition point for the partial architecture for R1. The partial architecture that results from this incremental composition is shown in Fig. 8.11.

8.3.3 Increment-3 - Requirement 3

R3: A customer must be able to deposit cash to the linked account that can be inserted into the cash slot. Approval must be obtained from the bank before physically accepting the cash.

Step-1: Extracting Elements of Component-based Systems from Requirements

For R3, first, the deposit and the accept cash computations verbs are identified; and an ordering execution is required to pass data between both of these computations. Next, the ‘before’ preposition that denotes ordering is extracted. As for data, the cash amount to be deposited and the approval status are identified.

5A PIPE is a specific case of a SEQ.
CHAPTER 8. DEFINING THE APPROACH

Step-2: Mapping of the Extracted Elements to X-MAN Elements

The extracted elements from R3 are mapped to X-MAN elements as listed in Table 8.3 (see the mapping result column). For R3, three components are assigned based on the extracted computations, namely AUT, AC and DP. For control, a PIPE and a guard adapter to evaluate the constraint data, i.e. the approval status, are mapped.

Table 8.3: Summary of Steps 1 and 2 for R3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Map to Component-based Constructs</td>
<td>Explanation</td>
</tr>
<tr>
<td></td>
<td>Extraction elements</td>
<td>Extraction category</td>
</tr>
<tr>
<td>Comp.</td>
<td>approval</td>
<td>Action noun</td>
</tr>
<tr>
<td></td>
<td>deposit</td>
<td>DT</td>
</tr>
<tr>
<td></td>
<td>accept</td>
<td>DT</td>
</tr>
<tr>
<td>Control</td>
<td>before</td>
<td>Preposition (ordering)</td>
</tr>
<tr>
<td></td>
<td>implicit constraint</td>
<td>[approved] guard</td>
</tr>
<tr>
<td>Data</td>
<td>approval</td>
<td>Data(constraint)</td>
</tr>
</tbody>
</table>

Step-3: Creating partial architectures

As the AUT component has already been mapped in the former increment, in this increment, in order to compose the partial architecture for R3 with the current system architecture (partial architecture for \{R1,R2\}), only the DP and the AC are modelled and composed. The result of this incremental composition is the partial architecture in Fig. 8.12(b). These elements are modelled as shown in Fig. 8.12(a).
8.3. A COMPLETE EXAMPLE: THE ATM SYSTEM

Step-4: Composing the partial architecture with the existing system architecture

The prior authentication has already been provided by the same AUT component in the partial architecture for \{R1, R2\}. Thus, the partial architecture for R3 is as shown in Fig. 8.12(a). The partial architecture can be incrementally composed to the existing system architecture, as depicted in Fig. 8.12(b).

\[\text{(a) Partial architecture for R3} \quad \text{(b) \{R1-R3\}}\]

\[\text{Fig. 8.12: Increment-3.}\]

8.3.4 Increment-4 - Requirement 4

R4: A customer must be able to make a transfer of money between any two accounts originated from the linked account.

Step-1: Extracting Elements of Component-based Systems from Requirements

From the fourth requirement, another ATM transaction that is able to transfer money from the linked account is extracted. In addition, the ‘linked account’ descriptive expression implies that an authentication is required in order
to perform the transfer transaction. For control, the ‘from’ preposition denotes ordering, whilst, for data, account information is extracted.

**Step-2: Mapping of the Extracted Elements to X-MAN Elements**

For each computation that is identified, a decision has to be made as to which component to assign the extracted computation. Later, the assigned component will be selected from the repository. For this requirement, the computation is assigned to a new component, which provides a transfer fund operation. A summary of both steps 1 and 2 is listed in Table 8.4.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extracted elements</td>
<td>Extraction category</td>
</tr>
<tr>
<td>Comp.</td>
<td>transfer DT</td>
<td>transferFund() computation » TF component</td>
</tr>
<tr>
<td></td>
<td>linked account descriptive expression</td>
<td>approve() computation » AUT component</td>
</tr>
<tr>
<td>Control</td>
<td>from Preposition (ordering)</td>
<td>PIPE (data dependency)</td>
</tr>
<tr>
<td></td>
<td>implicit constraint</td>
<td>[approved] guard</td>
</tr>
<tr>
<td>Data</td>
<td>approval implicit data store approval status (constraint)</td>
<td>constraint (HM-4)</td>
</tr>
<tr>
<td></td>
<td>account store account info</td>
<td>data (HM-4)</td>
</tr>
</tbody>
</table>

**Step-3: Creating partial architectures**

Based on the mapping result for R4, these elements are modelled (see Fig. 8.13(a)). We may observe that on top of the guard adapter, the solid lollipop interface is applied. The reason for this is that a guard adapter is not a composition connector.
and hence no open composition is allowed.

**Step-4: Composing the partial architecture with the existing system architecture**

The result of this composition can be illustrated in Fig. 8.13(b). In the existing system architecture, the AUT component has already been modelled. Hence, the only addition to the composition is the TF component together with the guard adapter that evaluates the approval status.

![Diagram](image1)

(a) Partial architecture for R4 (b) \{R1-R4\}

Fig. 8.13: Increment-4.

### 8.3.5 Increment-5 - Requirement 5

R5: A customer must be able to make a balance enquiry of the linked account.

**Step-1: Extracting Elements of Component-based Systems from Requirements**

From the fifth requirement, a balance enquiry computation is extracted. In addition, the ‘linked account’ descriptive expression denotes an authentication process, whilst the ‘of’ preposition indicates origin or source. Hence, the
preposition is used here as to denote ordering (see Table 8.5).

**Step-2: Mapping of the Extracted Elements to X-MAN Elements**

The balance enquiry and authentication computations are assigned to a Balance Enquiry (BI) and AUT components respectively. Meanwhile, the extracted control is mapped to a PIPE, whilst the approval status is assigned as constrained data. Since there is a constraint data, a guard adapter is required to evaluate the constraint, i.e. approval status.

<table>
<thead>
<tr>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>Extracted elements</td>
<td>Explanation</td>
</tr>
<tr>
<td>Comp. enquiry DT</td>
<td>HM-1</td>
</tr>
<tr>
<td>linked account</td>
<td>PIPE (data dependency)</td>
</tr>
<tr>
<td>approved</td>
<td>PIPE(AUT, [approved]+BI)</td>
</tr>
</tbody>
</table>

**Table 8.5: Summary of Steps 1 and 2 for R5.**

**Step-3: Creating partial architectures**

The mapping results of the extracted elements are modelled as in Fig. 8.14(a). Here, the control starts from the top and executes the AUT component, and continues with the connecting branch. The guard checks the approval status and if satisfied, invokes the computation in the BI component.
8.3. A COMPLETE EXAMPLE: THE ATM SYSTEM

Step-4: Composing the partial architecture with the existing system architecture

The result of this composition is illustrated in Fig. 8.14(b). In the existing system architecture, the AUT component has already been modelled. Hence, the only addition to the composition is the BI component together with the guard adapter that evaluates the approval status.

![Diagram](image)

(a) Partial architecture for R5  (b) {R1-R5}

Fig. 8.14: Increment-5.

8.3.6 Increment-6 - Requirement 6

R6: If the customer fails to be authenticated, the card will be rejected.

Step-1: Extracting Elements of Component-based Systems from Requirements

For the sixth requirement, the approve and reject card computations are identified. For control, the ‘if’ conjunction is extracted, whilst, for data, the customer and the approval status are specified.
Step-2: Mapping of the Extracted Elements to X-MAN Elements

These extracted computations are assigned to AUT and RC components respectively. The ‘if’ conjunction normally suggest branching execution. Nonetheless, there is no branching in such a case because only one component (based on its provided computation) will be selected. Hence, the conjunction is mapped to a guard adapter instead.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>authenticate DT</td>
<td>computation</td>
<td>AUT component</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reject DT</td>
<td>computation</td>
<td>RC component</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>implicit</td>
<td>Default ordering</td>
<td>PIPE(data dependency)</td>
<td></td>
</tr>
<tr>
<td>if Conjunction</td>
<td>implies selection execution (SEL or Guard)</td>
<td>パイプ(AUT, [approved(no)]→RC)</td>
<td>HM-3</td>
<td></td>
</tr>
<tr>
<td>approval constraint</td>
<td>[approved] guard</td>
<td>based on the constraint data</td>
<td>HM-4</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>customer</td>
<td>Data</td>
<td>store customer, card information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>approval constraint</td>
<td>(con-straint)</td>
<td>store approval status</td>
<td>constraint (HM-4)</td>
</tr>
</tbody>
</table>

Step-3: Creating partial architectures

As the authentication computation which is provided by the AUT component has already been modelled into the system architecture, the only component that is required to be composed is the reject card computation. A guard is also added to check the authentication status. These extracted elements can be modelled as in Fig. 8.15(a).

We may observe that, in contrast to the partial architectures in the previous
increments, this particular constraint for the guard is to evaluate for rejection approval status. This information is relevant for the refactoring task to be undertaken later.

Step-4: Composing the partial architecture with the existing system architecture

Fig. 8.15(b) demonstrates the outcome of this incremental step. Here, the additional constructs are the RC component together with a guard that evaluates the approval status. However, this time, the evaluation results into an invalid status.

8.3.7 Increment-7 - Requirement 7

R7: After each transaction, the ATM will display and print a receipt containing the transaction information.
Step-1: Extracting Elements of Component-based Systems from Requirements

In the following requirement, the display and print receipt computations are spotted.

Step-2: Mapping of the Extracted Elements to X-MAN Elements

These two computations are simplified, so as to be encapsulated in a single component. The reason for this is that these computations are performed by the same device. The extraction result column in Table 8.7 lists all the extracted elements of computations, control and data for R7. An abstract computation keyword that is the transaction data transformation is identified to represent all the relevant bank transactions.

Table 8.7: Summary of Steps 1 and 2 for R7.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>transaction* DT</td>
<td>conceptual computation »TR* component</td>
<td>HM-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>display DT</td>
<td>display() computation » PRT component</td>
<td>HM-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>print DT</td>
<td>print() computation » PRT component</td>
<td>HM-1</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>after Preposition (ordering)</td>
<td>PIPE HM-2</td>
<td>PIPE(TR, PRT)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>receipt, transaction</td>
<td>Data store receipt, transaction information</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

Step-3: Creating partial architectures

For the purposes of modelling, an abstract TR* component that represents all the bank transactions is modelled. In order to accommodate the print receipt
computation for each transaction, the existing composition needs to be extended by adding a new PIPE connector, so that once the transaction is completed, the information in the transaction can be piped and prepared to be displayed and printed. Otherwise, each bank transaction has to be modelled individually and later they have to be composed together to represent the similar effect. These extracted elements can be depicted as in Fig. 8.16(a).

Step-4: Composing the partial architecture with the existing system architecture

Fig. 8.16(b) depicts the result of this incremental step. Here, we compose the PRT component at the end of the composition point. This simulates that every time a transaction is executed, its detail can be displayed and printed.

8.3.8 Step-5: Finalising the system architecture

As the mapping process is performed incrementally, the options to be handled cannot be confirmed until the last requirement is dealt with, specifically relevant for a selection case. Hence, the SEL connector is not directly specified for this step in this example.
During this step, refactorings can be made to any parts of the architectural design, so as to improve the system diagram. In Fig. 8.16(b), we may observe that the guards are checking the same constraint values i.e., the approval status. In addition to this, another constraint for the chosen transaction type is also required. Both of these constraints can be encapsulated in a SEL connector, where the constraints can be checked first and the execution of the control will be based on the corresponding result. The final system diagram can be shown in Fig. 8.17.

As depicted in Fig. 8.17(b), a loop is added on top of the system diagram in order to simulate that the system is ready to be executed and will continue functioning when needed by the user. Otherwise, the system will only execute once, and has to be re-activated.

8.4 Validation of the Incremental Approach

The most common research strategy in SE solves some aspects of software development problems by producing a new procedure or technique and validating it by analysis or by discussing an example of its use [Sha02]. Hence, in order to validate the resulting system produced using this approach, a set of test cases
are created and assessed against the original set of requirement, and is considered as satisfying the requirement if it achieves the intended behaviours stated in the requirement. It is beyond the expectation of the intended purpose of this approach to achieve behaviours that are not explicitly stated in the requirements. To achieve this, however, an additional stage of analysis by the domain experts and the real users is required to discover the implicit requirements.

As the ATM example is a trivial example intended to demonstrate the formulation of the complete construction approach, we intend to apply a case study with a larger number of requirements for the validation purpose. This set of requirements has also been used as a reference example for component-based development [RRMP08].

8.4.1 The Trading System

The Trading System (COCOME) example is used for handling sales transactions in a supermarket. The system comprises 9 main functions covering (1) Process Sale, which handles Cash Desk operations; (2) Manage Express Checkout, which deals with transaction modes, i.e. normal and express; (3) Order Products, which allows the Store Manager to order products from suppliers; (4) Receive Ordered Products, which allows the Store Manager to update received orders; (5) Show Stock Reports, which permits the Store Manager to view all available stock; (6) Show Delivery Reports, which allows the Enterprise Manager to generate reports; (7) Change Price, which permits the Store Manager to update a product’s price; (8) Product Exchange (on low stock) Among Stores, which handles product orders between stores and (9) Remove Incoming Status, which allows the Store Manager to update the received product. Altogether, there are 47 requirements. These requirements are provided in Appendix C.1.1. The complete execution of the
COCOME using the incremental approach, is provided in Appendix C.

8.4.1.1 The COCOME System Architecture

Now we briefly discuss the complete COCOME system derived from the keywords we extracted from all the requirements for COCOME. We wish to demonstrate two important things: firstly that the encapsulation in our component model does indeed enable us to identify elements of component-based systems individually and separately from the requirements; and secondly that the system does indeed satisfy the requirements.

Fig. 8.18 shows the overall architecture of COCOME. It is a client-server architecture. Fig. C.46 shows the system in X-MAN derived from keywords extracted from the requirements.

![COCOME Architecture Diagram](image)

![Fig. 8.18: COCOME architecture.](image)

It models only the client side in Fig. 8.18. The system comprises two sub-architectures: Cash Desk and Store Client. Computations provided by the servers in Fig. 8.18 can be deemed as ‘remote’ computation units of components in the X-MAN system. For example, in Fig. C.46, computations of the INV component reside in the server. Interactions between servers in Fig. 8.18 are not modelled because they are not deterministic, e.g. interaction between Store Servers and Enterprise Server. Nonetheless, we have covered this in our implementation as a
8.4. VALIDATION OF THE INCREMENTAL APPROACH

separate system with its own execution thread and scheduler.

8.4.1.2 Implementation

Fig. 8.19 shows the composition in X-MAN for the COCOME system in Fig. C.46.

The figure illustrates the implementation of the complete COCOME system using our component model X-MAN. The top part of the figure demonstrates composition of the identified components, whilst the bottom part of the figure shows the Cash Desk execution console, which simulates the Cash Desk GUI. In our implementation, connectors are implemented as Java control structures e.g. a `SEL` as an “if...else” statement.

8.4.1.3 Validating the COCOME System

To validate the COCOME system we created, we executed a prescribed set of test cases presented in [RRMP08]. Using the final system, which is constructed based on the derived architecture (see Fig. C.46), we managed to successfully execute all of the following test cases.
CHAPTER 8. DEFINING THE APPROACH

TC-UC1A Purchase of goods with cash payment.

TC-UC1B Concurrent purchase of goods at more than one cash desk.

TC-UC1C Purchase of goods with card payment.

TC-UC1D Invalid item id read, manual entry of item id.

TC-UC1E Wrong PIN entry for credit card, card validation fails.

TC-UC3A Generate report of low stock product.

TC-UC4A Order low stock products, correct delivery is recorded.

TC-UC5A Generate report of available stock in a store (Store Manager).

TC-UC6A Generate report of cumulated available product in the enterprise (Enterprise Manager).

TC-UC6B Provide report containing mean time to delivery for each supplier (Enterprise Manager).

TC-UC7A Change price of a product (Store Manager).

TC-UC7B Search and display an item (Store Manager).

TC-UC8A Product exchange among stores.

Test Case for Use Case 1A

Fig. 8.20(a) and (b) show the execution of Test Cases for UC1A, which test for the purchase of goods with cash payment. The test case is considered passed once the item has been entered and the payment has been made. The system shall display the paid amount and the change amount to the Store Client.
8.4. VALIDATION OF THE INCREMENTAL APPROACH

(a) Test Case-1 A1  (b) Test Case-1 A2

Fig. 8.20: Test cases execution – TC-UC1A.

Test Case for Use Case 1B

This test case is dedicated to test the concurrent purchases of goods at more than one cash desk. In order to simulate this, we created two instances of the Store Server (SS) and performed the sale transactions. See Fig. 8.21a and Fig. 8.21b. Both of these executions are executed concurrently. These two cash desks processed sale transactions from different clients.

(a) Test Case-Use Case 1B Cash Desk 1  (b) Test Case-Use Case 1B Cash Desk 2

Fig. 8.21: Test cases execution – TC-UC1B.
Test Case for Use Case 1C

This test case tests for the purchase of goods with card payment without any exceptions. Given a valid password, the user shall be able to pay for the purchased items. See Fig. 8.22a.

(a) Test Case-Use Case 1C

Fig. 8.22: Test case execution - TC-UC1C.

Test Case for Use Case 1D

If an item id cannot be read, the system shall allow manual entry of the item id. See Fig. 8.23a.

Test Case for Use Case 1E

Fig. 8.23b shows the execution of a wrong PIN entry for credit card validation. The user shall be allowed to try as many times as required until he opt for cash payment.
8.4. VALIDATION OF THE INCREMENTAL APPROACH

Test Case for Use Case 3A

Fig. 8.24a shows the execution of Test Case-UC3A, which tests for generation of low stock products by the Store Manager. For this test case, the Store Manager also needs to be authenticated prior to the report generation. The test case passes if the report is generated.

Test Case for Use Case 4A

Low stock products shall be ordered and correct delivery shall be recorded. See Fig. 8.24b. Once the order is updated, the amount of the ordered product in the inventory will also be updated and the order details are removed from the order database.

Test Case for Use Case 5A

Fig. 8.25 illustrates the report generation for available stock in a store. This function is only available for Store Manager. The test case is considered passed when the report is able to be generated.
CHAPTER 8. DEFINING THE APPROACH

Test Case for Use Case 6A

The Enterprise Manager (EM) shall be able to generate report of cumulated available product in the enterprise (see Fig. 8.26a). The successful criterion for this test case is when the report is generated by the EM.

Test Case for Use Case 6B

In addition, the EM shall also be able to create report containing mean time to delivery for each supplier. The passing criterion for this test case is when the
8.4. VALIDATION OF THE INCREMENTAL APPROACH

report is generated by the EM (see Fig. 8.26b(b) and Fig. 8.26c(c)).

(a) Test Case-Use Case 6A  (b) Test Case-Use Case 6B1  (c) Test Case-Use Case 6B2

Fig. 8.26: Test cases execution - TC-UC \([6A, 6B]\).

Test Case for Use Case 7A

The Store Manager (SM) shall be able to change the price of a product (see Fig. 8.27a). The product must be identified first by searching for the product, and once the change is made, the product information is displayed to verify the changes.

Test Case for Use Case 7B

The Store Manager shall be able to search and display product information (see Fig. 8.27b).

Test Case for Use Case 8A

As we have mentioned in Section 8.4.1.1, we included the servers in the component’s implementation. Therefore, the inter-server communication between
Store Server and Enterprise Server is not modelled in the architecture shown in Fig. 8.18. That is why the execution of Test Case UC-8A, the product exchange among stores, is not tested based on this architecture.

The final COCOME system architecture after refactoring is illustrated in Fig. C.46.

### 8.4.2 Other Case Studies

Table 8.8 lists a summary of the applied, executed and documented case studies and examples in this thesis.

<table>
<thead>
<tr>
<th>Case studies/Examples</th>
<th>No. of requirements</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trading System (COCOME)</td>
<td>47</td>
<td>Medium</td>
</tr>
<tr>
<td>Steam Boiler System</td>
<td>30</td>
<td>Medium</td>
</tr>
<tr>
<td>Video Store System (VSS)</td>
<td>19</td>
<td>Medium</td>
</tr>
<tr>
<td>ATM System</td>
<td>7</td>
<td>Small</td>
</tr>
<tr>
<td>Word Count</td>
<td>5</td>
<td>Small</td>
</tr>
</tbody>
</table>

The second case study is that of the Video Store System (VSS), which is taken
from [LSB98]. The VSS provides and manages video rental transaction and customer record management. Altogether, there are 19 requirements, hence having the same number of increments. For the VSS, only one refactoring task was encountered, which was carried out after the final increment. Here, the purposes of refactoring were to restructure and combine parts of the duplicated architecture. This task is labelled in Fig. C.65, whilst the analysis of the refactoring is included in Section C.2.3. The complete execution of the VSS is demonstrated in Appendix C.2.

The third example is the Word Count, in which, given a file, the number of word will be counted and displayed. This simple example is drawn from [Sch06, p. 434]. In this example, we identified the pattern to refactor mutual exclusive guards and simplified them into a SEL. As mentioned in Chapter 7, most of the encountered refactoring cases whilst executing the case studies and examples are to simplify the architectural design. The discussion on this refactoring task for Word Count is included in Section C.3.4, while the complete execution of the Word Count example is provided in Appendix C.3.

In addition to the documented case studies and the examples, a number of case studies from the literature have also been experimented with. One of the non-trivial examples is the Steam Boiler case study [ABL96]. This example contains 30 requirements specifying the control of the water level in a steam-boiler. The system comprises the steam-boiler, a water level measurement device (WLMD), four water pumps, four pump controller devices (PCD), a steam measurement device (SMD) and a message transmission system (MTS). The program operates in several modes of operation: initialisation, normal, degraded, rescue and emergency stop. By using the ECF tool, the resulting final architecture is shown in Fig. 8.28, with 24 components and 11 composition connectors (with 6 guards).
8.4.2.1 Summary

In these case studies and examples, all the defined steps in the approach to constructing component-based systems have been demonstrated, applied and presented. The primary concern is to identify elements of component-based systems as defined in the X-MAN component model, namely computations, control and data. Following this premise, a complete COCOME example has been demonstrated as to apply all the heuristics and design decisions defined in this thesis (see Appendix C.1). It has also been elucidated that the identified computations, control and data can be used to guide us in constructing the complete component-based system. Although this approach is basically heuristic, and requires human guidance and decision making, we believe this is possible because the underlying component model provides a way to realise such an approach.

8.5 Issues and Discussion

The following list addresses a few anticipated issues and challenges during the adoption of the entire approach.
8.5.1 Requirements Problem

This approach does not claim to solve any requirement problems such as ambiguities, incompleteness or inconsistencies. These issues should be resolved and clarified with the customer. It is important to note that requirement problems are beyond the scope of this work. However, these issues may contribute to problems during the mapping or composition stages.

8.5.2 Element Extraction Exceptions

As mentioned in Section 8.2.3, when there is any missing keyword extraction during the element extraction step, this can affect the later steps. The possible causes of this scenario have been discussed. Whenever these issues occur, the designer may choose to skip the increment and deal with the requirement later.

8.5.3 Component Selection

The current component selection mechanism for this approach works on the basis of signature matching. The extracted computation keywords are assigned to a conceptual component (logical) and this component will be matched with an existing component in a repository. In this approach, the repository comprises pre-existing components (which have been developed), the generic composition connectors and adapters. Each time the construct is selected from the repository, an instance of the construct will be created and initialised during the run-time phase.

8.5.4 Validation of the Derived Architecture

A design that results in the satisfaction of the specification requires the relevant validation process. With the constraint of addressing the explicit requirements,
the applied testing technique can only be made based on the set of original requirements. The set of test cases will be executed against the derived architecture using this approach. An example of such testing is provided in Appendix 8.4.1.3.

8.6 Summary

This chapter has presented the entire approach to achieving the construction of component-based systems directly from raw requirements. This approach is based on the X-MAN component model, which supports the separation of control and computation. This separation feature, in turn, distinguishes the underlying fundamentals of what is to be extracted with other existing work in the literature. This approach is basically heuristic, and requires human guidance and decision making. Nonetheless, the steps and rules that are followed have been adequately defined for the sake of consistency. The following chapter presents evaluation and further discussion on the entire approach.
Chapter 9

Evaluation and Discussion

This chapter aims to demonstrate the validity of the approach to constructing component-based systems directly from NLR using the incremental composition mechanism. In order to perform such a demonstration, the succeeding section presents (1) a preliminary empirical validation of the incremental approach; (2) an analysis based on reflections and lesson learned of the case studies execution, as included in Appendix C; and (3) comparison with related work.

9.1 Introduction

In this thesis, an incremental approach to constructing component-based systems directly from NLR is presented. In essence, such an approach allows us to (1) systematically map the extracted keywords directly to their corresponding executable architectural constructs; (2) deal with any number of requirements; and therefore, it should scale up to arbitrarily large requirement documents. Once the approach is formulated, the next stage is to validate the derived architecture (system).

Three strategies are adopted in pursuing the validation of the incremental
approach. Firstly, a preliminary empirical validation of the incremental approach is executed and reported. Secondly, based on the analysis of the executed case studies, the experiences and lesson learned are discussed. These analysis in the form of reflections, is deliberated to highlight the properties of the approach. Finally, a comparison with BE is included to demonstrate the differences with this approach. The following section starts with the first strategy.

9.2 Preliminary Empirical Validation of the Incremental Approach

This section describes a preliminary experiment to validate the incremental approach. It starts by laying out the objectives and setting of the experiment. The remainder of this section is mainly divided into two parts; the first part covers the pilot study, whilst the second part includes the main experiment. In each of these parts, the analysis, results and discussion are presented. Finally, a summary of the experiment is included at the end of the section.

9.2.1 Objectives

The experiment aimed to investigate the proposed incremental approach to constructing component-based systems directly from natural language requirements (NLR). This experiment is a preliminary empirical validation of the proposed incremental approach. Nonetheless, due to the limitation of employing industrial professionals as subjects, we employed computer science and information system undergraduate and postgraduate students, and academic staff as subjects in this research. It has been a common practice in research to use students as subjects [Ber04]. They were trained sufficiently well to perform the given tasks
9.2. PRELIMINARY EMPIRICAL VALIDATION

as recommended in [Tic00].

The goal definition for the experiment is listed below:

Object of the study: The incremental approach

Purpose: To undertake a preliminary experiment to validate the proposed incremental approach to constructing component-based systems directly from NLR.

Focus: The focus of the experiment are:

1. To evaluate the effectiveness of the incremental approach in terms of correctness.

2. To investigate the application of the heuristics in analysis and making design decisions during each incremental step.

Perspective: From the point of view of the users of the incremental approach.

Context: In the context of users with software engineering background.

9.2.2 Research Questions

Research questions are the focus of the study, and are established from research objectives [VST+09, Jar00]. According to Collis and Russey [CH03], research questions can be viewed from positivist and phenomenological studies. Research questions in the positivist study generally express relationships between variables. On the other hand, research questions in the phenomenological study often evolve during the process of research and may need to be refined as the study progresses [CH03]. In addition to research questions, propositions are used to identify and focus on the issues to be investigated in a research. The collected experiment data
is then compared to the propositions in order to support or reject the propositions [RH09].

The following research questions (RQ) and propositions (P) have been addressed in this experiment:

RQ-1: What is the level of correctness when using the incremental approach in constructing component-based systems directly from NLR?

Based on RQ-1, the propositions are:

P1-1: The element extraction process of the incremental approach will produce the architectural elements to construct component-based system.

P1-2: The extracted architectural elements based on the extraction process can be incrementally mapped and modelled to construct component-based system.

RQ-2: Are the expected heuristics being applied when making design decisions during element extraction and mapping of the extracted elements to the XMAN model?

Based on RQ-2, the propositions are:

P2-1: The extraction heuristics will be applied when making decisions in identifying the relevant architectural elements.

P2-2: The mapping heuristics will be applied when making decisions in mapping the extracted elements into XMAN modelling elements.

9.2.3 Instrumentation and Materials

In the initial stage of the experiment, a consent letter and an invitation letter were distributed to the candidate participants. The invitation letter described
information pertaining to the experiment and highlighted the experiment’s main purpose.

In addition, the training materials for the (1) Fundamentals of the XMAN semantics, (2) the element extraction and (3) the mapping processes; (4) the Extractor tool; and (4) the Exogeneous Component Framework (ECF) tool, which had been prepared by the researcher, were also distributed for reference to the participants.

Nonetheless, the participants were reminded that the main intention of the experiment was to validate the incremental approach, and not to focus or be biased towards the provided tools. Without these tools, it would be difficult to concretely experiment the approach. However, we had no intention of claiming that these tools are efficient or easy to use, with regard to the incremental approach.

9.2.3.1 Analysis of the Effectiveness of the Incremental Approach in terms of Correctness

The interpretation criteria for measuring correctness of the propositions (1) P1-1 that is based on the correctness of the extracted elements compared to the extraction benchmark in each increment; and (2) P1-2 is based on the accumulation of the correctness and syntax scoring values compared to the set of benchmarks for each increment. Based on the interpretation criteria for the correctness of proposition P1-1, scoring values were calculated based on the scoring of extraction outcomes. The ideal outcomes for each requirement from the extraction process were set as the benchmark. The benchmark was derived from the outcomes of each incremental step in Sec 8.3. In that section, all the expected architectural elements were set as benchmark for the experiment task.

An evaluation characteristic was incorporated into the process of evaluating
the quality of the extraction outcomes. Another aspect to be considered for assessing the correctness of the implemented solution in the experiment was based on the analysis of the result of the incremental design. These results were benchmarked against (1) a set of expected element extraction results; and (2) a set of expected designs for each increment. Although it is acknowledged that a design should be a creative process based on the elements that are needed to be modelled, we were concerned with how the composition was decided.

For each of those items (1) and (2), a scoring range of 0-2 was used. Each result was evaluated and given a score of 2 marks, if the derived incremental outcome was (1) accurate as the benchmark; or (2) the extraction outcome was not exactly as the benchmark, but was highly relevant. Meanwhile, 1 mark would be given if the extraction result was not as accurate as the expected benchmark. This case includes (1) only one of the extracted element categories was correct, which means either one of the computation or control was correct; (2) the extracted elements were correct, but not arranged in the right ordering; or (3) the extracted elements were not completely correct, and not arranged in the right ordering.

Finally, a 0 mark was assigned if the extraction result completely deviates from the benchmark. The relevant scenarios were when (1) both the extracted element categories were wrong; or (2) no useful elements were extracted; or (3) extraction was not solely based on the corresponding requirement; or (4) extraction was decided not based on the provided heuristics, which led to incorrect and unjustifiable design decision. In addition to the above cases, a 0 mark would also be given if the participant extracted a single computation only, or the outcome was incomplete to derive a valid composition. Table 9.1 lists the summary of scoring criteria for the extraction process.

Meanwhile, for the mapping and modelling process, the outcomes of partial architecture design in each increment would be studied and analysed. The design
elements would be assessed based on two criteria: (1) the correctness of the modeled elements and (2) the syntax of the notation used to model the extracted elements. For the correctness criteria, each design was given a score of 2 marks for an accurate representation. This includes (1) the composition of the components and connectors were logically correct and accurate as the benchmark; or (2) the composition was not exactly as the benchmark (but could still be considered as logically correct) based on the extracted elements.

On the other hand, a 1 mark was given for cases (1) an inaccurate representation, but not completely wrong, based on the extracted elements; (2) the composition could be considered logically correct based on the extracted elements; or (3) the composition could be considered logically correct, but not completely based on the extracted elements. In the latter case, participants might add additional architectural elements which were not derived from the extraction process. Finally, a 0 mark was specified for any incorrect representation including cases where (1) violated the previously incremented composition, for example changed or removed previous incremental design; (2) the composition was wrongly modeled; or (3) the ordering or selection execution was incorrectly represented. As may be seen from Table 9.2, summary of correctness scoring criteria for the mapping process is provided.
In addition, for the second criteria i.e. in view of syntax, each design outcome was given a score of 2 marks for cases where (1) valid representation using the XMAN syntax by using valid composition points and valid composition connectors or adapters; or (2) use of sequential execution connector i.e. PIPE instead of SEQ or vice versa, but could still represent the incorporated behaviours from the NLR.

A 1 mark was assigned for an incomplete syntax representation including (1) correctly model only one of the extracted computations or control; or (2) the partial architecture is logical but inappropriate use of vertical or horizontal composition, which may change the previously incremented design; or (3) used valid composition points but missing some features. Finally, a 0 mark would be given for violating the XMAN syntax representation, including cases where (1) composition was done at invalid composition points; or (2) model was not designed based on the extracted elements for either one or both the computations and control. The summary of the mapping scoring criteria in terms of syntax is listed in Table 9.3.

### Table 9.2: Summary of Mapping Scoring Criteria for Correctness

<table>
<thead>
<tr>
<th>Score</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>the composition of the components and connectors were logically correct.</td>
</tr>
<tr>
<td></td>
<td>the composition was logically correct based on the extracted elements.</td>
</tr>
<tr>
<td>1</td>
<td>an inaccurate representation, but not completely wrong based on the extracted elements.</td>
</tr>
<tr>
<td></td>
<td>the composition could be considered logically correct based on the extracted elements, but was not accurate.</td>
</tr>
<tr>
<td>0</td>
<td>the composition could be considered logically correct, but not completely based on the extracted elements.</td>
</tr>
<tr>
<td></td>
<td>violated the previously incremented composition.</td>
</tr>
<tr>
<td></td>
<td>the composition was wrongly modeled.</td>
</tr>
<tr>
<td></td>
<td>the ordering or selection execution was incorrectly represented.</td>
</tr>
</tbody>
</table>
9.2. PRELIMINARY EMPIRICAL VALIDATION

Table 9.3: Summary of Mapping Scoring Criteria for Syntax

<table>
<thead>
<tr>
<th>Score</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>valid representation using the expected XMAN syntax by using valid composition points and valid composition connectors or adapters. used sequential execution connector i.e. PIPE instead of SEQ or vice versa, but could still represent the incorporated behaviours.</td>
</tr>
<tr>
<td>1</td>
<td>correctly model only one of the extracted computations or control. logical partial architecture, but use of vertical or horizontal composition which might change the previously incremented design. used valid composition points but missing some features.</td>
</tr>
<tr>
<td>0</td>
<td>composition was done at invalid composition points. model was not designed based on the extracted elements for both the computations and control.</td>
</tr>
</tbody>
</table>

9.2.3.2 Analysis of the Application of the Extraction and Mapping Heuristics

Based on RQ-2, the propositions are (1) P2-1: The extraction heuristics will be applied when making decisions in identifying the relevant architectural elements; and (2) P2-2: The mapping heuristics will be applied when making decisions in mapping the extracted elements into XMAN modelling elements. For both the propositions, survey items were used and the responses were compared with the benchmarks provided by the researcher. The benchmark comprises a set of expected extraction and mapping heuristics for each of the incremental steps (that is for each requirement).

Analysis of the application of the extraction and mapping heuristics were based on a set of questionnaire items. This questionnaire consisted of 14 items to represent each of the extraction and mapping processes for each requirement.\(^1\) For each of the requirements in each increment, participants were asked to identify the relevant heuristics that they adopted. An example for an increment of a requirement can be shown in Fig. 9.1. This figure is an excerpt of a questionnaire item for application of element extraction heuristics. In addition, Fig. 9.2 depicts

\(^1\)http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Requirements.html
CHAPTER 9. EVALUATION AND DISCUSSION

Fig. 9.1: An example of a Questionnaire Item for Extraction Heuristics

Q1. Requirement-1: The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).

**TASK 1: ELEMENT EXTRACTION (Refer Appendix 1)**

1. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?

   a. Identifying computations
   b. Identifying computations from user interaction
   c. Identifying conceptual component
   d. Identifying control
   e. Identifying data
   f. NONE, explain (__________________________)

The results of the experiment for the extraction and mapping processes were analysed and evaluated. For the extraction process, all applied heuristics answers were benchmarked against the expected set of answers. Based on the accumulated number of responses from the data for each answer i.e. for each requirement, the average number of responses was calculated. This value determined whether the participants had applied the expected heuristics when making design decisions.

Fig. 9.2: An example of a Questionnaire Item for Mapping Heuristics

TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS (Refer Appendix 2)

2. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?

   a. Map control flows to composition connectors
   b. Choose between PIPE or SEQ
   c. Map control to a GUARD
   d. Combine or separate connectors
   e. Combine or separate components
   f. Computation inferences
   g. Handling data elements
   h. No indication of any computation or control
   i. NONE, explain (__________________________)

   (__________________________)

The results of the experiment for the extraction and mapping processes were analysed and evaluated. For the extraction process, all applied heuristics answers were benchmarked against the expected set of answers. Based on the accumulated number of responses from the data for each answer i.e. for each requirement, the average number of responses was calculated. This value determined whether the participants had applied the expected heuristics when making design decisions.
during each increment or whether they were using other sources of knowledge or their own experiences in identifying those elements.

9.2.4 Pilot Study

The purpose of the pilot study was to validate the instruments to be used in the experiment. The pilot experiment was conducted at the Faculty (Kulliyyah) of Information and Communication Technology (KICT), International Islamic University Malaysia (IIUM) during Semester 1, 2012/2013. Participants for the pilot study were three Master students at the faculty. In order to use the approach, the participants were required to acquire the essential knowledge. Hence, prior to the experiment, the participants were trained on (1) the fundamentals of the XMAN semantics, the element extraction and the mapping processes; (2) the element extraction heuristics; (3) the mapping heuristics; (4) the Extractor tool; and (5) the Exogeneous Component Framework (ECF) tool. Duration of the training was approximately 1 hour and 30 minutes.

9.2.4.1 Demographic Profile

Participants of the pilot study were recruited on a voluntary basis. The pilot study involved three participants, who were pursuing their master study at the faculty. All of them had background in general systems analysis and design, however, only one participant declared to have experience in modelling or developing component-based systems. Compensation was given to the participants for their participation in the experiment.
9.2.4.2 Experimental Setup

The pilot study took place according to the availability of participants in a computer lab in the faculty. The training session lasted for approximately 1 hour and 30 minutes for the pilot session. Any confusion was clarified during the Q&A session before the experiment started. The participants were then briefed on their tasks for the experiment. Participants were allocated another additional maximum period of 3-hour to complete the given tasks, including a brief demographic survey at the beginning of the session.

At the beginning, the participants were asked to fill in the demographic survey questionnaire. See Appendix B, Set A. Then, the training was conducted by the researcher. After the training, the participants were briefed on the experiment tasks. In the pilot study, the participants were given a set of NLR statements taken from the ATM example\textsuperscript{2}. This ATM example was chosen on the basis of the effort and time required in executing and solving the requirements. In addition, the approach had to be constrained to be executed incrementally, which means only one requirement should be dealt with at a single time and added to the existing design until they reach the final solution. The intended aim was to derive a system design using the XMAN component model; hence the effort to enhance the design was not considered.

Subsequently, while executing the tasks, they were required to identify the adopted extraction or mapping heuristics by using a set of questionnaires. See Appendix B, Set B. In the end, the participants were asked to fill in the post-experiment questionnaire. For this task, they were asked to rate the difficulty level of the training session and the experiment tasks using a five point Likert scale {‘Very Easy’, ‘Easy’, ‘Average’, ‘Difficult’, ‘Too Difficult’}. See Appendix

\textsuperscript{2}http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Requirements.html
For each of the participants, an assistant was assigned to assist them in assessing the heuristics used throughout the experiment. The reason behind this was that it would be cumbersome to execute the tasks and at the same time provide the required feedback for each incremental step. The assistants were also trained prior to the experiment. Hence, during each incremental step, the assistants clarified and recorded the heuristics adopted by the participants.

The participants were reminded to (1) represent the extracted elements into the XMAN component-based elements according to the given training materials and not to be concerned with design issues; (2) incrementally compose the existing design with additional behaviours. Whilst performing the extraction and mapping processes, they were expected to clarify their design decisions with regard to the adoption of any relevant extraction or mapping heuristics. The reason was mainly to investigate the heuristics’ usage level in assisting them in the incremental approach. Without these heuristics, design decisions might be based on personal knowledge or experiences, hence design decisions might be the result of unjustifiable design decisions.

9.2.4.3 Result of the Pilot Study

This subsection provides the findings of the pilot study, which starts with the result of the effectiveness of the incremental approach, and continues with the result of the application heuristics whilst using the incremental approach.

a. Result of the Analysis of the Effectiveness of the Incremental Approach

This subsection describes the results of the analysis of the effectiveness of the incremental approach. The analysis of the correctness of the extraction process
CHAPTER 9. EVALUATION AND DISCUSSION

is described. The results from participants were assessed against the benchmark and score values were assigned accordingly. In understanding the scoring criteria, refer Table 9.1. Table 9.4 lists the scoring values for the extraction process for the pilot study.

Table 9.4: Extraction Scoring for Pilot Study

<table>
<thead>
<tr>
<th>User/Increment</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-01</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>User-02</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>User-03</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>Average</td>
<td>78.57%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, most of the participants extracted either the same outcomes as the benchmark or a set of highly useful architectural elements from the NLR. This statement is supported by the high scoring values for R1, R2, R3 and R7, which contributed to more than 50% of the requirements with high extraction scores. In contrast, the common mistakes in the extraction process were cases where (1) only one of the extraction category was correct (i.e. computation or control); (2) the extraction was not completely correct, and not arranged in the correct ordering; (3) extraction was not solely dependent on the current requirement; and (4) extraction was incomplete to compose valid composition. In fact, during the trainings, the participants were reminded to derive extraction elements, which should be useful in the later stage. In addition to detailed extraction scoring, an average value was calculated to represent the overall correctness score for all the participants. The average correctness score for extraction was 78.57%. This value showed that in average, 78.57% of the extraction scoring were correct as compared to the benchmark.

In interpreting the correctness criteria for mapping process, the outcomes were assessed based on (1) correctness of the logical representation; and (2) syntax of
the modelling elements. The first correctness characteristic related to the level of correctness of the design outcomes compared to the benchmark prepared by the researcher. This case included assessment in terms of logical representation of the extracted elements according to XMAN fundamentals. The detailed cases for the correctness characteristics are provided in Table 9.2. The second correctness characteristic involved the level of correctness in terms of syntax of the representation that used the XMAN fundamentals. This case assessed the design outcomes from the perspective of valid notation used to represent the corresponding behaviours. The detailed cases for this characteristic are listed in Table 9.3.

The scoring values for the mapping of the extracted elements to XMAN model was elaborated based on two interpretation criteria. Firstly, the scoring was based on the correctness of its logical representation. Mapping outcomes from each increment that were derived using the ECF tool were printed and analysed. Each of the design from each increment were assessed using the benchmark and score values were assigned. Table 9.5 provides the scoring values for each participant. The total score value was calculated and the total average score was derived from the total score, which was 64.29%. The average value represents the correctness of logical representation of all the participants as compared to the benchmark. The average value was high considering not all the participants had any background on XMAN fundamental except for the given training.

Table 9.5: Summary of Correctness Scoring Value

<table>
<thead>
<tr>
<th>User/Increment</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-01</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>User-02</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>User-03</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64.29%</td>
</tr>
</tbody>
</table>

Based on Fig. 9.3, a high percentage of correctness scores were derived from R1, R2 and R7. These values corresponded to the logical representation of the
extracted computations and control from the requirements, which achieved the same level as the benchmark. Nonetheless, the most significant negative outcome was R5 for which none of the participants got the correct score. The justifications for this particular increment were based on illogical representation, or breaking the previous composition. In the latter case, this inherently violates the encapsulated behavior represented in the previous increments.

Fig. 9.3: Summary of Correctness Scoring Value

The second interpretation criterion for correctness of the mapping outcomes was in terms of the syntax of the representation. The same set of procedures as in the logical assessment was adopted in an attempt to assess the syntax of the derived design. Nonetheless, the scoring characteristics were relevant to assess the syntax of the design outcomes. See Table 9.3. Each of the derived design outcomes was evaluated and given a score value. This means that the design should adhere to the modelling syntax defined in XMAN model. The following Table 9.6 provides a summary of scoring value for the syntax.

Based on Fig. 9.4, we may observe that the syntax used for R1 and R2 were considered accurate according to XMAN notation and used valid composition points for incremental composition. The issues with R4, R5 and R6 might be
9.2. PRELIMINARY EMPIRICAL VALIDATION

Table 9.6: Summary of Scoring Value for Syntax

<table>
<thead>
<tr>
<th>User/Increment</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-01</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>User-02</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>User-03</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69.05%</td>
</tr>
</tbody>
</table>

related to (1) use of invalid composition points; (2) not being modelled based on the extracted elements for any of the extraction categories i.e. computation or control; or (3) valid composition syntax, but used an invalid composition point.

Fig. 9.4: Summary of Syntax Scoring Value

On average, the correctness value for the mapping process, considering the logical representation (64.29%) and syntax (69.05%) was 66.67%. These findings were included to support the P1-1 and P1-2 propositions that the incremental approach helped in assisting participants to (1) extract relevant architectural elements and (2) map the extracted elements to XMAN elements i.e. computations and control.
b. Result of the Analysis of the Application of the Extraction and Mapping Heuristics

Each survey item was analysed against a set of expected answers, which acted as a benchmark. From the pilot study, the items could be grouped into two categories i.e. for extraction and mapping processes. Based on the survey items, participants were asked to assess which heuristics they adopted whilst performing each of the processes. For example, during the first increment, participants were asked to identify which heuristics were applied during the extraction of the component-based elements from the given requirement statement. These sets of answers were checked against the expected answer scheme sets and were summarised as presented in Table 9.7.

<table>
<thead>
<tr>
<th>Question</th>
<th>RESULT-A (%)</th>
<th>Question</th>
<th>RESULT-B (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.00%</td>
<td>2</td>
<td>55.56%</td>
</tr>
<tr>
<td>3</td>
<td>66.67%</td>
<td>4</td>
<td>58.33%</td>
</tr>
<tr>
<td>5</td>
<td>66.67%</td>
<td>6</td>
<td>77.78%</td>
</tr>
<tr>
<td>7</td>
<td>77.78%</td>
<td>8</td>
<td>33.33%</td>
</tr>
<tr>
<td>9</td>
<td>66.67%</td>
<td>10</td>
<td>50.00%</td>
</tr>
<tr>
<td>11</td>
<td>88.89%</td>
<td>12</td>
<td>22.22%</td>
</tr>
<tr>
<td>13</td>
<td>77.78%</td>
<td>14</td>
<td>66.67%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>74.21%</strong></td>
<td><strong>TOTAL</strong></td>
<td><strong>51.98%</strong></td>
</tr>
</tbody>
</table>

The reason why we had odd numbered questions for extraction and even numbered questions for mapping was that for each increment, participants were asked to identify heuristics for extraction of the particular requirement. Subsequently, from the extracted elements, they were required to identify the mapping heuristics for the same requirement.

Fig. 9.5 presents the average percentage values of Result-A, which were achieved from the use of extraction heuristics during the extraction process, while Fig. 9.6 shows the average in percentage values for Result-B, which was contributed to the
9.2. PRELIMINARY EMPIRICAL VALIDATION

It is apparent from Table 9.7 that there was a significant difference of usage for both of these set of heuristics. The first value was derived from the average of the total scores of survey items relevant to the extraction heuristics. The participants were asked to identify which heuristics that they adopted when making decisions...
about the extracted architectural elements. The application of extraction heuristics contributed 74.21% compared to the expected benchmark. In addition, participants were also asked to identify which mapping heuristics were applied when they mapped the extracted element to XMAN syntax. However, the application of mapping heuristics only contributed 51.98% of the expected benchmark. This difference might occur because of lack of knowledge of the XMAN semantics and the incremental composition concept.

A side by side comparison of the application of extraction and mapping heuristics is shown in Fig. 9.7. It is interesting to observe that there were cases of high application of extraction heuristics, but low application of the expected mapping heuristics. Further research might need to be done to investigate the correlation between those properties.

These findings nonetheless support the proposition P2-1 that the participants applied the relevant elements extraction heuristics to extract the relevant architectural elements from NLR. Although the findings for mapping heuristics were low compared to the scoring of the extraction heuristics, there is nonetheless evidence to support proposition P2-2 that the participants applied the mapping
9.2. PRELIMINARY EMPIRICAL VALIDATION

The heuristics to map the extracted elements to XMAN elements i.e. computations and control.

9.2.4.4 Result of the Post-Pilot Study

After executing the experiment, participants were asked to rate the provided training in terms of its difficulty level, ranging from too easy to too difficult. The trainings were divided into five parts that were (1) the XMAN fundamentals; (2) the element extraction process; (3) the Extractor tool; (4) the mapping of the extracted elements to XMAN elements; and (5) the ECF tool. All the participants of the pilot study had mixed feelings when they were asked to rate the difficulty levels of the provided training. Nonetheless, all of the perceived feedback was rated as average and below. This showed that they perceived so as to they understood the training sessions of the experiment.

In addition, they were required to rate the tasks during the experiment. The tasks involved were (1) extraction of the candidates of the relevant architectural elements; (2) identification of extraction heuristics; (3) analysis of the interaction; (4) modelling composition using XMAN; (5) identification of mapping heuristics; and (6) handling incremental composition. For the pilot study, in general, most of the participants rated the modelling task using XMAN notation as the most difficult task. This perception can be deemed as predictable because it was their first-hand experience of learning and using the XMAN fundamentals and notation.

9.2.4.5 Discussion

The pilot study was successful in proving the propositions P1-1 and P1-2 in an attempt to address RQ-1. The level of correctness of the incremental approach was addressed by using correctness interpretation criteria for both the extraction
and mapping processes.

In addition, RQ-2, which investigates the application of extraction and mapping heuristics, was achieved through support from propositions P2-1 and P2-2. In summary, we could conclude that the participants applied the relevant heuristics whilst making judgment in the extraction and mapping processes. It is also worth noting that without these heuristics as guidance, it is difficult to formulate an objective interpretation as we are dealing with subjective nature of the NLR.

It is also worthwhile being concerned about the constraints during the experiment such as limited training duration prior to the execution of the experiment; no complete example provided; too many tasks to be performed including the use of two tools i.e. The Extractor and the ECF tools. The participants were asked to use the Extractor tool for the extraction process and then use the ECF tool to map and model the extracted elements. While using the tools, they were asked to capture the screen shots and save the screen shots into a Word processing file. In addition, they were also required to identify the relevant heuristics that they applied when they extracted elements and mapped the extracted elements to XMAN notation. Considering these required efforts, it was difficult to focus throughout the specified duration.

From the pilot study, a questionnaire item was updated in terms of wording and organisation. In the demographic survey during the pilot study, the participants were asked to indicate the number of years of experience related to IT industry. In order to organise the data, instead of directly stating the number of years, a set of range to indicate the property is added. This way, it is easier to code the data for the main experiment. See Appendix B, Set A.
9.2.5 The Main Experiment

The main experiment was conducted using the same setting and procedure as the pilot study, which was performed at a computer laboratory in the faculty. It was organised into two sessions to suit the availability of the recruited participants. Following the same procedures as in the pilot study, prior to the experiment, the participants were given the same training as the pilot study. Duration of the training was approximately 50 minutes. The training duration was slightly shorter than the training session for the pilot study because the training content had been simplified to cover relevant knowledge for the participants.

9.2.5.1 Demographic Profile

A total of 11 participants were recruited on a voluntary basis. Among them, four were male and seven were female participants. In addition, 3 participants hold PhD degree, 6 participants hold MSc and 2 participants hold Bachelor degree in relevant IT or computer science fields. All of them had the required background in general systems analysis and design. Most of them also had working experiences in IT industry, but only two participants stated that they had specific experience in component-based software engineering. The experiment was held over two sessions to suit the availability of the participants. The start and end time were recorded; and in average, each participant spent approximately 3 hours and 7 minutes for the experiment excluding the training. Compensation was paid to all the participants for their effort and time during the experiment.

9.2.5.2 Instrumentation and Materials

The experiment replicated the instrumentation and materials from the pilot study, which had been updated after the execution of the pilot study.
9.2.5.3 Experimental Setup

The experiment took place according to the availability of participants in a computer lab in the faculty. In order to suit the availability of the participants, two sessions were arranged. Both the training sessions lasted for approximately 50 minutes. Any confusion was clarified during the Q&A session before the experiment started. The participants were then been briefed on their tasks for the experiment. The remaining procedures for the main experiment follow the same procedures in the pilot study.

In the pilot study, for each of the participant, an assistant was assigned to aid the participants in identifying the heuristics used throughout the experiment, while the participants were solving the extraction and mapping processes. Nonetheless, due to the constraint of assistants’ availability, we were not able to assign assistants to each of the participants. However, three assistants including the researcher were present during the whole execution of those two sessions of the experiment. All of these assistants took turns to assist the participants. We believe that this situation would not significantly affect the result of the experiment. Nonetheless, having assistants could help to reduce the required effort.

In this main experiment, the participants were also given the same a set of NLR statements\(^3\). The participants were also reminded to (1) represent the extracted elements into the XMAN component-based elements according to the given training materials; and (2) incrementally compose the existing design with additional behaviours. Whilst performing the extraction and mapping processes, they were expected to clarify their design decisions with regard to the adoption of any relevant extraction or mapping heuristics. Additionally, the participants were also reminded that the main intention of the experiment was to validate

\(^3\)http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Requirements.html
the incremental approach, and not to be biased on the Extractor and ECF tools. This is in line with the intention of the experiment. Apart from this, it is also important to state that none of the participants in the pilot study was involved in the main experiment.

Given the basic structure of the experiment, and having considered the setting and participants background, the experiment is now presented in detail.

9.2.5.4 Analysis

The analysis for the main experiment replicated the analysis in the pilot study. See Section 9.2.3.1 and Section 9.2.3.2.

9.2.5.5 Result of the Main Experiment

This subsection lays out the results of the main experiment for (1) the effectiveness of the incremental approach in terms of correctness; and (2) the application of heuristics in extraction and mapping processes.

a. Result of the Analysis of the Effectiveness of the Incremental Approach

Scoring values were used as the interpretation criteria for measuring the correctness of the propositions (1) P1-1 that is based on the correctness of the extracted elements compared to the extraction benchmark in each increment; and (2) P1-2 that is based on the accumulation of the correctness and syntax values compared to the set of benchmarks for each increment. First, the result of the extraction scoring for the experiment is presented.

Table 9.8 lists the scoring values for the extraction process. Interestingly, the total average score of the extraction as compared to the benchmark was 83.77%. This value showed that in average, most of the participants extracted either the
CHAPTER 9. EVALUATION AND DISCUSSION

accurate outcomes as the benchmark or highly set of useful architectural elements from the NLR.

Table 9.8: Extraction Scoring

<table>
<thead>
<tr>
<th>Increment</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-01</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>78.57</td>
</tr>
<tr>
<td>User-02</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>71.43</td>
</tr>
<tr>
<td>User-03</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>92.86</td>
</tr>
<tr>
<td>User-04</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>13</td>
<td>92.86</td>
</tr>
<tr>
<td>User-05</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>User-06</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>85.71</td>
</tr>
<tr>
<td>User-07</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>85.71</td>
</tr>
<tr>
<td>User-08</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>User-09</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>User-10</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>92.86</td>
</tr>
<tr>
<td>User-11</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>100</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>18</td>
<td>20</td>
<td>17</td>
<td>18</td>
<td>17</td>
<td>19</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>83.77%</td>
<td>83.77%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above statement is evidenced by the high scoring values for all the requirements. Based on Fig. 9.8, the chart shows the average extraction scoring values for each participant. The common mistakes with extraction process based in the result, were related to (1) only one of the extracted elements category was correct; (2) the extracted elements were useful, but not arranged in the correct ordering; or (3) the extracted elements were not completely correct, and not arranged in the correct ordering. In addition, amongst other identified errors were (1) not using the suggested extracted elements; and (2) extraction were incomplete for further composition. In general, the low score values were the result of the common mistakes identified.

The second part is an analysis of the interpretation criteria for the correctness of the mapping processes. The scoring values for the mapping of the extracted elements to XMAN model would be elaborated based on two interpretation criteria. Firstly, the scoring was based on the correctness of its logical representation. Mapping outcomes from each increment that were derived using the ECF tool
were printed and analysed. Each of the design from each increment was assessed using the benchmark and score values were assigned. The detailed scoring cases for the correctness criterion is provided in Table 9.5. Table 9.9 provides the scoring values for each participant. The total score value was calculated and the total average score was derived from the total score that is 70.13%. The average value was high considering all the participants did not have any background on XMAN fundamental except for the given training.

Based on Fig. 9.9, a high percentage of correctness scores were derived from
R1, R2, R3 and R7. These values corresponded to the logical representation of the extracted computations and control from the requirements, which achieved high scoring value compared to the benchmark. In contrast, lower score values were caused by either one or combination of these cases (1) illogical representation; (2) breaking the encapsulation of the previous composition; (3) the ordering of the composition was wrong.

![Fig. 9.9: Result of Correctness Scoring Value](image)

The second interpretation criterion for the correctness of the mapping outcomes is in terms of syntax. The same set of procedures as in the pilot study was adopted to validate the syntax of the derived design. Each of the derived design outcomes were evaluated and given a score value. Refer to Table 9.6. In general, the produced design should adhere to the modelling syntax defined in XMAN model. The following Table 9.10 provides the detailed scoring value for syntax based on each increment for each participant.

Based on Fig. 9.10, we can observe that syntax used for R1, R2, R3 and R4 were considered high because of the scoring value, which contributed to more than 70%. Overall, in average 76.62% of the representations were accurate according to the XMAN notation and used valid composition points for incremental
composition. The issues with the rest of the increments may relate to (1) not modelled based on the extracted elements for any one of the extraction categories i.e. computation or control; (2) valid composition syntax, but used invalid composition point; (3) used invalid composition points; or (4) not modelled using the extracted elements. These observations were gathered based on the scoring assessment. Each of the design outcomes was analysed and given a scoring value with its justification.

![Fig. 9.10: Scoring Value for Syntax](image-url)
On average, the correctness value for the mapping process considering the logical representation (70.13%) and syntax (76.62%) was 73.38%. These findings are included to support the P1-1 and P1-2 propositions that the incremental approach helps in assisting participants to (1) extract relevant architectural elements and (2) map the extracted elements to XMAN elements i.e. computations and control.

b. Result of the Analysis of the Application of the Extraction and Mapping Heuristics

Based on RQ-2, the propositions are (1) P2-1: The extraction heuristics will be applied when making decisions in identifying the relevant architectural elements; and (2) P2-2: The mapping heuristics will be applied when making decisions in mapping the extracted elements into XMAN modelling elements. For both the propositions, survey items were used and the responses were compared with the benchmarks provided by the researcher.

Each survey item was analysed against a set of expected answers, which acted as a benchmark. From the pilot study, the items were grouped into two categories i.e. for extraction and mapping processes. Based on the survey items, participants were asked to identify which heuristics that they adopted whilst performing each of the processes. These sets of answers were compared with the expected benchmark sets and are summarised in Table 9.11.

Fig. 9.11 presents the average percentage values of the Result-A, which were achieved for the use of extraction heuristics during the extraction process, while Fig. 9.12 shows average in percentage values of the Result-B, which were contributed to the use of the mapping heuristics during the mapping process. These values indicated whether the use of the provided heuristics was significant or not. Without these heuristics, participants might have to rely on their experiences and
Table 9.11: Adoption of the Extraction and Mapping Heuristics

<table>
<thead>
<tr>
<th>Extraction</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESULT-A</td>
<td>RESULT-B</td>
</tr>
<tr>
<td>Q1 77.27%</td>
<td>Q2 57.58%</td>
</tr>
<tr>
<td>Q3 90.91%</td>
<td>Q4 60.61%</td>
</tr>
<tr>
<td>Q5 75.76%</td>
<td>Q6 48.48%</td>
</tr>
<tr>
<td>Q7 90.91%</td>
<td>Q8 51.52%</td>
</tr>
<tr>
<td>Q9 100.00%</td>
<td>Q10 38.64%</td>
</tr>
<tr>
<td>Q11 93.94%</td>
<td>Q12 45.45%</td>
</tr>
<tr>
<td>Q13 90.91%</td>
<td>Q14 45.45%</td>
</tr>
<tr>
<td><strong>TOTAL AVG</strong> 88.53%</td>
<td><strong>TOTAL AVG</strong> 49.67%</td>
</tr>
</tbody>
</table>

It is apparent from Table 9.11 that there is a significant difference in use of background knowledge in order to perform the extraction and mapping processes.

Fig. 9.11: Application of the Extraction Heuristics

Fig. 9.12: Application of Mapping Heuristics
both of these set of heuristics. The value for average Result-A was derived from the average of the total scores of survey items relevant to extraction heuristics. The participants were asked to identify which heuristics that they adopted when making decisions about the extracted architectural elements. The application of extraction heuristics contributed 88.53% compared to the expected benchmark. In addition, participants were asked to identify which mapping heuristics were applied when they map the extracted element to XMAN syntax. However, the application of mapping heuristics (Result-B) only contributed 49.67% of the expected benchmark. The difference may occur because of lack of knowledge of the XMAN semantics and the incremental composition concept.

A comparison of the application of extraction and mapping heuristics can be shown in Fig. 9.13. It is interesting to observe that there were cases where high application of extraction heuristics, but low in application of the expected mapping heuristics. Further research may need to be done to investigate correlation between those properties.

![Fig. 9.13: Comparison between Extraction and Mapping Scoring](image)

These findings, even so, support the proposition P2-1 that the participants
applied the relevant elements extraction heuristics to extract the relevant architectural elements from NLR. Although the findings for mapping heuristics were quite low, nonetheless, there is an evidence to support proposition P2-2 that the participants applied the mapping heuristics to map the extracted elements to XMAN elements i.e. computations and control.

9.2.5.6 Result of the Post-Experiment

After executing the experiment, participants were asked to rate the provided training in terms of its level of difficulty, ranging from too easy to too difficult. The trainings were divided into five parts that were (1) the XMAN fundamentals; (2) the element extraction process; (3) the Extractor tool; (4) the mapping of the extracted elements to XMAN elements; and (5) the ECF tool. In general, most of the participants perceived the trainings as average in terms of level of difficulty. See Table 9.12.

<table>
<thead>
<tr>
<th>Training material</th>
<th>Too Easy</th>
<th>Easy</th>
<th>Average</th>
<th>Difficult</th>
<th>Too Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMAN Fundamentals</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Element Extraction</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>The Extractor Tool</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Mapping to Component based elements</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>The ECF Tool</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In addition, the participants were required to rate the tasks during the experiment. The purpose of this item was to get feedback on their perception of the difficulty levels of each of the given task. Based on Table 9.13, most of the participants, collectively, rated the extraction of architectural elements, identifying extraction heuristics, modelling task using XMAN notation, identifying mapping heuristics and handling incremental composition as quite difficult tasks. This perception could be deemed to be common because it was their first-hand experience in learning the XMAN fundamentals and notation. Amongst all the tasks,
only analysing the interaction of the extracted element task (labelled as Task C in Table 9.13) was perceived as average, in terms of the level of difficulty.

<table>
<thead>
<tr>
<th>TASK</th>
<th>TOO EASY</th>
<th>EASY</th>
<th>AVERAGE</th>
<th>QUITE DIFFICULT</th>
<th>DIFFICULT</th>
<th>TOO DIFFICULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

A: Extracting candidates of component-based system.
B: Identifying which extraction heuristics is relevant for a specific requirement.
C: Analysing the interaction (behaviour).
D: Modelling the composition using XMAN.
E: Identifying Which mapping heuristic(s) is to be applied for a particular extraction.
F: Handling incremental composition.

9.2.5.7 Discussion

In line with the results from the pilot study, the main experiment was successful in proving the following propositions P1-1 and P1-2 in the attempt to address RQ-1. The level of correctness of the incremental approach was addressed by using correctness interpretation criteria for both the extraction and mapping processes. In addition, RQ-2, which investigated the application of extraction and mapping heuristics among the participants, was achieved through support from propositions P2-1 and P2-2. The evidence from both the propositions concluded that the participants applied the relevant heuristics whilst making judgment in the extraction and mapping processes. It is also worth highlighting that without these heuristics as guidance, it is difficult to formulate an objective interpretation as we are dealing with subjective nature of the NLR.

In addition, future research might also be relevant to investigate the correlation between extraction and mapping outcomes. From the initial finding, it was
found that there was slight differences in the correctness of the mapping outcomes as compared to the correctness of the extraction outcomes. Nonetheless, it is interesting to note the reasons that might lead to this finding.

9.2.6 Threats to Validity

One of the potential threats to the internal validity of the experiment relates to the adoption of benchmarking. The use of benchmarking in Software Engineering (SE) empirical research is also recommended as stated in [WKP10]. Although benchmarking has its own risks in terms of difficulties, the authors believe that benchmarking should be a valid baseline for SE research [WKP10]. In this experiment, a set of benchmarking items was used for the assessment correctness criteria as the scoring value of the extracted elements and the derived design. Nonetheless, the scoring value is hugely assessed by the researcher and hence, may lead to bias. However, a set of detailed scoring assessments was provided in order to support an objective assessment in an attempt to reduce biasness.

Another potential threat to the experiment arises from the background of the participants. Most of them were not familiar with component-based systems. Nonetheless, the training was supposed to be sufficient for them to derive the expected architectural design incrementally.

In addition, the small number of participants is a further potential threat to the validity of the experiment. A small sample was chosen because of the expected difficulty of obtaining participants. According to [CH03], the aim of qualitative research in a phenomenological paradigm is to gain deeper insights, and it is possible to conduct research using small number of sample. On average, participants in this experiment spent 3 hours and 7 minutes on the experiment, excluding the training sessions prior to the experiment. It is worth noting that
all the participants were selected on a voluntarily basis.

9.3 Summary

The experiment presented herein showed the preliminary validation of the incremental approach to constructing component-based systems directly from NLR. The findings supported the propositions with regard to the correctness of the extraction and mapping processes and also the application of both the extraction and mapping heuristics. We believe that further investigation which relates to correlation of the extraction and mapping processes could be relevant to a better understanding of design decisions made by human analysts.

9.4 Analysis of the Case Studies

The resulting incremental approach, detailed in Chapter 8, is validated and documented using two case studies and examples\footnote{Including the ATM example\cite{Sch06} provided in Chapter 8.} intended to show the applicability of the approach. The documented case studies, which are provided in the Appendix C, are the Trading System (COCOME) \cite{RRMP08}, the Video Store System (VSS) \cite{LSB98} and the Word Count \cite[p. 434]{Sch06} examples. The complete executions of each of the case studies are provided in the same appendix. These case studies capture the application of all the heuristics and design principles presented in the corresponding chapters, the Incremental Composition in Chapter 4, the Element Extraction in Chapter 5, the Mapping of the Extracted Elements in Chapter 6, and the Architecture Refactoring in Chapter 7.

The following section furnishes the strategy towards validity of the approach,
which is obtained from experiences of the execution of the case studies and examples using the approach.

9.5 Reflections on the Approach

This section presents the reflections, as a result of the execution of the case studies and examples, in four main viewpoints: (1) the effects of the intrinsic properties of the X-MAN component model; (2) the constraints of the requirements; (3) the effects of the adopted features in this approach; and (4) the effects of the supporting tools. Each of these viewpoints will be broken down into specific issues to justify the impacts.

9.5.1 Properties of the X-MAN Component Model

The X-MAN component model, as presented in Chapter 3, offers three benefits, mainly from the aspects of component encapsulation, compositionality and reusability. Here, each of these aspects will be reflected on.

The Effects of Encapsulation

The keyword extraction process is carried out for a single requirement at a time. This is possible because of encapsulation in the X-MAN model. It is desirable because analysing a single requirement is more manageable than the usual practice of analysing all requirements together. It is also beneficial because it scales to any number of requirements, and because it is always a finitely terminating process. Although this approach is basically heuristic, and requires human guidance and decision making, the steps and rules have been adequately defined for the sake of consistency.
In addition, the extracted keywords are mapped only once, and the mapping is confined to a single component or connector. This gives evidence that the keywords can be extracted and mapped individually and separately, without having to be constrained with the dependencies with the rest of the architecture elements. This is important because it makes the whole extraction and mapping process much easier to manage, and it enables the system architecture to be scalable according to any number of requirements.

As a consequence of this effect, it allows us to realise the mapping process and visualise the architectural constructs in the architecture diagram. Fig. 9.14 illustrate this concept, in which keywords of control and computations are extracted from a piece of requirement based on the heuristics and design decisions, with the help of the Extractor tool. These keywords are then mapped to the X-MAN elements and these elements are used in constructing the component-based system architecture.

A more specific example shown in Fig. 9.15, which contains a sample of the keywords extracted from the requirements. They are keywords extracted from the requirements for the Sale Transaction process. The table shows clearly each extracted keyword, the requirement from which it was extracted, the component
9.5. **REFLECTIONS ON THE APPROACH**

or connector in the X-MAN system (see Fig.C.46) it was mapped to, and the label of this component or connector. For example, the keyword ‘enterItemID’ was extracted from requirement [UC1-R3], mapped to the component BarCodeScanner (BCS); the keyword ‘or’ was extracted from requirement [UC1-R3], and mapped to the connector SEL in the architecture. The detailed extraction process is included in Section C.1.11.

![Fig. 9.15: Keywords Extraction for Sale Transaction.](image)

This table shows clearly that each keyword is extracted from just one requirement. Moreover, it is mapped only once, and mapped to either a component or connector (adapter). This gives evidence that keywords can be extracted and mapped individually and separately. This is the essential feature that makes the extraction and mapping process manageable and scalable to accommodate any number of requirements.

Despite the claim that the lack of external dependencies in the X-MAN component model is presented as an advantage, some may argue this claim with respect to data dependencies. The X-MAN model mainly deals with control, and it is assumed that data always follow control. For single-threaded execution only.

---

5For single-threaded execution only.
viewed as *black-box* components which are *truly* independent.

**The Effects of Compositionality**

The X-MAN model defines how components can be systematically composed in order to build a larger composition. By adhering to the composition rules motivated by the semantics of the model, components are properly defined in an anticipated well-formatted piece of software. As a result, the structure of the architecture will be inherently explicit and systematic. Based on the results of the case studies and examples, the derived architectures have clear structures with an explicit separation of concerns, namely control and computation. Since the model offers the separation of control and computation encapsulated in separated architectural elements, this makes the extraction from natural language more credible and plausible in most of the cases.

**The Effects of Reusability**

The reusability feature can be achieved in both the design and deployment phases. During design, components and connectors are built and deposited into a repository. In design, the main aim is to maximise reusability, whilst in deployment, the main concerns are to retrieve the existing components and connectors from the repository, and compose them for a specific application. In view of this approach to constructing component-based systems, we apply the model only in the deployment phase because the systems being built are for specific applications, not for generic systems in contrast to the design phase.

When composing partial architectures, extracted computations, which are assigned to components, are then selected from a repository. We assume these components already exist in the repository. Issues pertaining to component selection are assumed to be dealt with by using the signature matching mechanism.
When the required component is not found, it must be created in a separate life-cycle component development and stored in the repository.

With regard to approaches that attempt to map requirements to architecture (see Chapter 2), the resulting architectures in most of the approaches are design specifications. These specifications are either created via an abstraction process or a series of refinement processes. Once the final architecture is derived, the design will be passed to the developer team for construction. In contrast, as an effect of the reusability property, the resulting architecture is an executable specification which directly relates to an executable system. In fact, it is worth mentioning that at any single point of increment, the derived architecture is already an executable system. Furthermore, in general, the reusability factor leads to reducing effort, time and cost in contrast to constructing a new system from scratch.

9.5.2 Requirement Issues

In the mission to address the gap between requirements and architecture, it is worthwhile to highlight some of the concerns pertaining to these requirement issues. The following sections discuss and present the issues with regard to the requirements.

Types of Requirement

As specified in Chapter 1, this approach is confined to accommodate functional requirements only, leaving the quality or non-functional requirements (NFR) un-handled. This decision does not intentionally mean to stress the importance of functional requirements over NFR; representing behaviours in the architectural elements itself is an open research issue on its own. Furthermore, the chosen component model covers behavioural properties, and does not handle non-functional
properties at the point of this writing.

The element extraction process presented in Chapter 5 has sufficiently addressed what can and cannot be extracted from functional requirements. In brief, statements that only contain data, relationships, facts, rules, or constraint information obviously do not add on to the behavioural property of the architecture being built. Nonetheless, relevant information needs to be properly documented.

Problems in Natural Language Requirements

Despite the fact that NLR may contain problems such as ambiguity, inconsistency and incompleteness, it is mostly still inevitable to use NLR in the early stage of the requirements engineering process [BK00]. Some believe that NLR shall never be completely replaced by formal specification [Rut77], and recommended accommodating them instead. This approach to constructing component-based systems directly from NLR in a way seconded and addressed this recommendation.

Change of Requirements

From the context of requirement changes, we believe this approach can be useful. Although the execution of the case studies and examples do not explicitly validate this claim, with the features of incremental composition and architecture refactoring, we believe that requirements changes can be attained. Another piece of supporting evidence is that the ordering of requirements is not an issue, as long as the designer is able to identify valid composition points to be applied. Hence, the incremental composition and architecture refactoring mechanisms can be used to support changes in requirements.
9.5.3 The Effects of the Selected Properties of the Incremental Approach

This thesis has provided the foundations of the approach to constructing component-based systems directly from NLR. In conjunction with this, there are two significant properties that embrace this approach: (1) incremental composition and (2) architecture refactoring. This section provides insights into the effects of these features.

The Effects of Incremental Composition

The ability to derive an architecture incrementally while guaranteeing the incremented behaviours are maintained has a significant effect. The substance of incremental is the fact that it can address scalability. When progressing with small increments, the architecture can be built up and, eventually, the final architecture can be derived. Moreover, the scalability issue can be addressed as the approach is designed to accommodate any number of requirements. Secondly, the incremental construction of architecture/system also supports scalability with the ability to support behaviour preservation of the incremented architecture.

In the semantics of the X-MAN, without the incremental composition, we need to deal with increasing hierarchies as the number of connectors are added. Such a strategy hinders the smooth transition during the design and also adding considerable efforts to the approach. By allowing open composition points during each increment, this enables more components to be composed with the existing composition connectors, instead of adding new connectors for each increment.
CHAPTER 9. EVALUATION AND DISCUSSION

The Effects of Architecture Refactoring

The promising advantages of refactoring are: (1) to reduce architectural design errors, and (2) to provide systematic reuse of design knowledge and proofs [MQR95, TB01]. The same advantages are sought in this approach. The architecture derived in each increment has already been validated and, hence, no rework effort has to be spent whenever the changes to the architectural structure are made.

Both the incremental composition and refactoring raise concerns in regards to behaviour preservation, but each of these mechanisms has their own significant context. In architecture refactoring, behaviour preservation mainly addresses the preservation of the incremented behaviours while dealing with structural changes. These changes are due to the need to restructure the design in order to (1) simplify them or (2) allow further composition. These needs are not directed towards incremental composition.

9.5.4 The Effects of the Tools Support

This approach provides two supporting tools, namely the Extractor (see Sections 5.5 and 8.2.2.1) and the Exogeneous Composition Framework (ECF) (see Section 8.2.5.1) tool. Each of these tools offers support from different contexts. The following sections reflect on each of these tools.

The Extractor Tool

The Extractor tool supports the approach by (1) highlighting the extracted textual analysis tags according to the categories of keywords extraction (e.g. verb, noun, control); (2) automatically removing part-of-speech taggers that are irrelevant (e.g. articles) and (3) providing a simple documentation for the extracted elements, and this can be used as a reference for the succeeding steps.
Without this tool, the systems analyst or designer has to elicit the information from each requirement manually. Despite the use of the tool, some may argue about its effectiveness, when even by using the tool, the user still needs to read the statement. Nonetheless, the tool helps by highlighting the selected categories of the extraction, thus reducing mental effort in identifying the component-based elements, even with a relatively trivial help. We do not in any way claim that by having this tool, the extraction process can automatically be undertaken.

The Exogeneous Composition Framework Tool

The Exogenous Composition Framework (ECF) tool aids in the modelling and composition of the partial architectures using the incremental composition mechanism. This tool helps by automatically checking the composition rules according to the semantics of X-MAN. Thus, this can support the reduction of other non-essential constraints during composition, and the user’s effort could be particularly centered on the composition of the partial and system architecture.

The significance of the ECF tool is, without doubt, the fact that it enables the designer to construct systems and then execute them. It also allows us to validate a final system architecture with respect to the system’s requirements. The execution of the test cases can show that the system satisfies its requirements. The success of the tool in general also experimentally validates the approach of the system construction directly from the requirements.

The next section presents the second strategy towards validation of the approach. Here, a comparison study with related work, namely the BE approach, is provided.
9.6 Comparison with Related Work

Majority of the investigated approaches (as discussed in Chapter 2) adopt abstract representation of requirements modelling, either by using procedural or object-oriented paradigm. The focus of these extractions or mapping processes, thus, are grounded on the foundations of the underlying abstraction of these paradigms.

As laid out in Chapter 2, among the investigated approaches that claimed to have addressed the gaps between requirements and architecture, the only work that is closely related to this approach is the BE\textsuperscript{6} approach. The main reason is that BE deals directly with each requirement by building behaviour trees (BT). These BT are then integrated to build a design behaviour tree (DBT) during the integration process. Recently, BE has been extended by incorporating a design stage claiming to provide a scalable methodology that allows \textit{end-to-end} process covering from requirement to design [Mye10]. In the BE component model, components are defined based on hardware components (see [Mye10, p.100-103]). The model's characteristics include the followings:

1. components encapsulate computations;

2. a component can have multiple configuration settings;

3. computations are separated from configurations;

4. flow of control determines configurations;

5. a component can easily be redesigned to make a new component;

6. a component can interface between two incompatible components.

\textsuperscript{6}Information on BE can be accessed at url: http://www.behaviorengineering.org/
As a result of BT integration, a design behaviour tree (DBT) is produced. The component model introduces a behaviour run-time environment (BRE) that minimises communications between system and components [Mye10, p.116-117]. This BRE takes the DBT and processes it into a deployment composition tree (DpyCT) as deployable component-based designs.

We consider the architecture in BE as being built once, without any increment or refinement step. Despite having a clear-cut strategy in dealing with design models, the BE methodology has not intended to incrementally produce the architecture. To date, BE has not put focus on behaviour preservation, as it is not required when constructing the architecture in such a strategy. The concern in BE is more into reducing complexity when dealing with requirements. That is why, the methodology used in BE highlights the use of BT in order to achieve the aim. In contrast to our approach, the architecture is obviously constructed in each increment. Furthermore, this architecture is an executable architecture, as it is linked directly to executable components being retrieved from the repository.

9.7 Issues and Discussion

This section provides the relevant issues and discussion from the view of support for software development project, requirements authoring styles, potential effects of requirements ordering, dependencies and prioritisation, satisfying requirements and the derived architecture using the incremental approach.

9.7.1 Support for Software Development Project

In product line system development, domain requirements are identified and classified into commonality and variability features to achieve reusability [LYZZ06].
These features are selected and composed to build systems in product line system development. Products comprise common and variable features, all of which can be set accordingly. Based on our knowledge, there is no work in product line system development that is geared towards building systems incrementally. The existing incremental concern might be relevant to the production of different combination of these features for each product in the same product line.

Nonetheless, it is potentially useful to adopt the incremental approach in order to support the development of product line systems. The main justification is supported by the fact that product line systems are constructed using pre-existing software parts. This is in line with the principle proposed in the incremental approach so as to be able to produce product line systems systematically and hierarchically. An important consideration with regard to the incremental approach which makes the approach plausible is the underlying notion of the XMAN semantics that computations and control are totally separated. Hence, the incremental nature is grounded in the existence of exogeneous control, which eventually build the hierarchies of control structures. Without this concern, it is almost impossible to apply the incremental approach in product line system development.

On the other hand, the proposed incremental approach could also potentially work with Agile methodology. The main difference between Agile software development methodology with non-Agile (some refer to as plan-driven project) is its focus on being adaptable to change [NNG+07]. Agile’s main philosophy is to produce small intervals in each increment and further build that increment to a larger scale. The particular point worth considering is the notion that Agile’s principle is to produce working software over documentation [TKW03, KS04]. In light of achieving Agile’s aim, software developers are more focused on producing actual software artifacts i.e. coding than producing unnecessary documentation.
Agile methodology is also intended to be generic to suit various software development methods and techniques. It does not specify or fix what kind of techniques or tools should be adopted in any software project. Nonetheless, the essence of Agile is mainly to achieve small working increments, rather than putting effort into producing an exhaustive list of documentations. Agile-based projects mostly gained their agility from the expertise and experiences of the involved team members [Coc02, Boe02]. In addition, there is also a risk of dealing with irrecoverable architectural mistakes during the execution of the short cycles in Agile. This may be due to the shortcomings of the knowledge and experiences of the team members, whilst maintaining rapid changes in the implementation in each cycle [Boe02].

Merely by virtue of the general Agile philosophy, we can recommend the incremental approach to be adopted in an Agile setting. The main justification is that it recommends the software developers to produce small increments and inherently build systems incrementally. This is in line with our aim in the incremental approach of being able to produce partial architecture in each increment and compose it to the system design, which eventually leads to the final system architecture. Although we are aware that this is not the only concern in Agile, it is of interest to investigate the adoption of Agile in developing component-based systems. Regardless of whether or not Agile can best be suited to being adopted for the incremental approach, further investigation is indeed required. It would be interesting to know the implications of adopting Agile for the incremental approach.
9.7.2 Requirements authoring styles

According to Lausen [Lau02], styles of functional requirements can be categorized into human/computer interaction style, context diagram style, event and function list style, feature-style, screen and prototype style, task description style (user view) and scenario (use-case) style (system view). Lausen also states that the way requirements are stated differs for a number of factors, mainly (1) the notation used, (2) the level of ease of verification and validation purposes, (3) whether the requirements specify the product functions and its surroundings, or (4) whether the requirements simply identify what they do and its details. In reality, functional requirements specification may be created with a combination of these styles.

With regard to the functional requirements style classification, the input to the incremental approach defined in this thesis restricts only to textual-based functional requirements. That is, it may comprise human/computer interaction style, event and function list style, feature style, task description style and scenario (use case) style. Such a restriction implies that in order to maximize the use of the approach in the elements extraction process, requirements, which are stated using these styles, have higher probability of being discovered. In addition, inferences have to be made whenever no explicit component-based elements (computations, control and data) are available.

Nonetheless, we acknowledge the existence of different and also combinatorial requirements styles, with respect to the way in which the requirements are written. For instance, the use of use case specification might improve the effectiveness of the approach because use case specification provides the flow of event(s) according to the required ordering. To an extent, if the NLR is organized in such a way that ordering is already being considered, this will promisingly give an
impact to the proposed incremental approach. Nonetheless, further investigation is required to validate and provide evidence of such a case.

9.7.3 Potential Effects of Requirements Ordering, Dependencies and Prioritisation

In addressing the potential effects of requirements ordering in the incremental approach, one of the relevant issues is the concerns of its effects on the composition. The main aim to achieve whilst having increments is to preserve the previously incremented architecture. According to [MVG06], each transformation (i.e. refactorings) in model-driven development preserves behaviours of the source model while the structure is modified in the target model. Nonetheless, the incremental composition supports composition as well as decomposition at any increment, as described in Section 7.3.1. This feature facilitates both top-down and bottom up approaches and hence, the ordering of requirements can be addressed.

It is worth noting that most of the time, compositions of components are done using a bottom-up approach. This means that by default, the normal approach will be to extract the architectural elements from NLR, to select the relevant pre-existing components from the repository and to design the composition using XMAN semantics using a bottom-up approach. However, whenever requirements are not in the correct ordering, the designer shall still be able to construct the composition using the horizontal and vertical refactoring concepts, which are introduced in Section 7.2.

The only exception that may occur with regard to the requirements ordering is when there are requirements errors originating in the source of requirements. For instance, the stated requirements are incorrectly or inconsistently specified. These kinds of requirement errors are beyond the scope of this work and should
be handled accordingly before the requirements are used as an input to the incremental approach.

Requirements dependencies

Many researchers have acknowledged that requirements are typically not independent on each other [GV02, ZMZ05, ZMZ06], and various types of dependencies exist among them and between relevant software artefacts [RJ01, Egy03, Ram98, KBV12]. One of the critical success factors for software projects is to understand the dependencies and correlations between the underlying attributes [GV02]. Because of these dependencies, there is a considerable interest in investigating traceability techniques, which are relevant in capturing and identifying the relationships between these artefacts [RJ01, Egy03, Ram98, GF94, Got92, ACDL99, IMD05, MXP05, PS05, Boh95, HJD11]. The traceability concern is another challenging and non-trivial task to consider in understanding, capturing, managing and maintaining the dependencies among software artefacts.

The context of the incremental approach is in the deployment phase, which means that the designer will be using the approach to build a component-based system, and not to create components for reuse purpose. See Section 3.2 for the component life cycle. In the incremental approach, requirements might have dependencies, but in the current state of the work, the dependencies and their issues are not thoroughly addressed. When extracting a particular computations or control from NLR, the extracted elements will be mapped to architectural elements in the component (and connector) repository. Consequently, a computation is mapped to a candidate component, based on its specification (because we cannot see what is inside a black-box component). If the same or similar computation occurs in part of another requirement later on, the same procedure will take place.
This is justifiable because the incremental approach itself is relevant only in the deployment phase. During deployment, the aim is to develop a system by using pre-existing components and not being concerned to maximise reuse such as in the design stage.

**Requirement prioritisation**

The need for requirement prioritisation has long been recognised in the literature [Wie03, Sid96, Fir04, TCBB09]. Most of the needs of requirement prioritisation in the literature are rooted in issues such as constraints on resources, conflicting preferences amongst stakeholders, market opportunities, risks, product strategies, and costs [KTR+07, Fir04]. In order to select the correct set of requirements, decision makers should make an effort to understand the relative priorities of the requested requirements [Wie03]. After selecting the prioritised requirements, software developers may address those requirements within the allocated constraints before handling non-prioritised requirements.

Amongst the techniques for prioritising requirements in the literature are (1) *absolute* according to importance e.g. essential, conditional or optional [IEE98], mandatory, desired or best value ([HJD11, Som07]); (2) *relative*, which is more subjective according to the human analyst. Both techniques aim to release the planning of incremental software development. For a thorough review of these and other prioritisation techniques, see Karlsson [KTR+07], Lehtola and Kauppinen [LK04], Berander [BS09] and Moisiadis [Moi02].

In [TCBB09, Fir04], requirement prioritisation techniques are classified into:

1. clustering approaches: categorise requirements into different groups;

2. consensus-based approaches: geared towards getting consensus among stakeholders;
3. multi-criteria ranking: combine multiple relevant criteria into a single value [GV02];

4. pair-wise comparison: compare all requirements and identify their values [KTR+07];

5. voting systems: express preferences by the stakeholders [BS09];

6. financial approaches: based on financial measures or cost-value [KWR98].

As mentioned earlier, prioritisation is concerned with producing product releases by concentrating on requirements based on certain aspects of preference. Nonetheless, be it of any of those requirement prioritisation technique, the incremental approach should be useful in incorporating the selected set of functional requirements. The crucial feature of the incremental approach is the ordering of the behaviours stated in each of the requirement statements. The representation of the execution in XMAN is very much dependent on the ordering of behaviours that need to be captured.

9.7.4 Achieving requirements satisfaction

The central aim in software development, in general, is to satisfy user requirements. In this approach, we simply adopted and applied an existing black-box testing technique. This decision is based on the justification that components, as defined in the X-MAN are black-box components, thus, the testing technique that is applicable is the black-box testing. The test cases are derived from the external interactions via the top-level connector. Accordingly, the test cases cover the system execution paths only, and not the computations themselves because the components are black-box and cannot be accessed directly. We assume the
components are already fitted for composition, which means they are heretofore tested and ready to be deployed.

9.7.5 The Resulting Architecture

The resulting architecture produced by individual designers may be ranging from totally to slightly different designs. Regardless of the differences, the bottom line is that this approach provides a systematic process for reducing the abstraction process made during the analysis and design stages. Without such a process, the designers need to apply the abstraction process based on their own expertise and judgment. This may even lead to a profound difference with regard to the resulting architecture.

9.8 Summary

This chapter has provided the evidence of the validation of the approach to constructing component-based systems from NLR. The preliminary empirical validation has been executed and findings have been reported. Moreover, the reflections of lessons learned during the execution of case studies have been synthesised for the discussion. All of these outcomes from the analysis of the reflections and the comparison study have been imparted in the process of validating the approach. Finally, a comparison study with BE and some issues and discussion have also been provided.
Chapter 10

Conclusions and Future Work

In this thesis, the foundations for constructing component-based systems directly from NLR using incremental composition and supported by heuristics and design decisions have been defined. Central to this approach is a novel method, comprising these foundations based on the semantics of the X-MAN component model. This final chapter outlines the contributions and limitations of the research, provides further discussion and present recommendation for future work.

10.1 Research Contribution

This section relates all the research objectives as listed in Chapter 1 with their corresponding contributions in the respective chapters. Fig. 10.1 summarises the research contribution made in this thesis.

The analysis and review of the existing approaches with regard to handling the transition from requirements to system architecture has been presented in Chapter 2. Based on the analysis, the knowledge gap has been addressed. To the best of our knowledge, there is no existing approach that deals with each requirement, and at the same time incrementally builds an executable architecture.
10.1. RESEARCH CONTRIBUTION

The second objective, namely the formulation of the proposed approach, has been achieved in Chapter 8. This chapter consolidates and illustrates the entire idea of the approach, from extracting elements of the component-based systems to the creation of the system architecture. The heuristics for identifying and extracting elements of component-based systems have been covered in Chapter 5. This chapter provides the required guidance in the extraction process. In addition, Chapter 6 provides the links between the extracted elements and the corresponding X-MAN constructs.

The incremental composition and architecture refactoring features have been set out in Chapters 4 and 7. In these chapters, the required design decisions have been addressed in order to allow smooth transitions between the requirements and architecture stages. By adopting a textual analysis technique, the Extractor tool, which aids in the discovery of the elements of the component-based systems, has been developed. The tool has been presented in Chapter 5, Section 5.5.
CHAPTER 10. CONCLUSIONS AND FUTURE WORK

The third objective, that is to validate the proposed approach, is achieved through (1) feasibility analysis, (2) reflections from the executed case studies and examples and (3) a comparison with related work. An evidence that this approach works is provided by the execution of a complete ATM example (see Section 8.3). In addition, the execution of the provided case studies have also contributed to this objective (see Appendix C). Finally, a comparison with related work, namely the BE approach, has also been included.

10.2 Limitations and Discussion

An approach for constructing component-based systems directly from raw requirements has been presented. The primary concern here is to derive an architecture that satisfies all the requirements. Clearly the architecture that this method produces may not be the best possible design according to various criteria. For example, the corresponding system may not be efficient in terms of execution speed.

This approach is based on a specific component model that supports incremental composition, which again distinguishes this work from existing related approaches. It has been demonstrated how the approach works for incremental system construction using individual requirements. Such an incremental approach allows us to deal with any number of requirements, and therefore it should scale up to arbitrarily large requirements documents. To demonstrate that this is the case, a tool for assisting in the element extraction process, namely the Extractor tool and another tool for modelling, that is the Exogeneous Composition Framework (ECF) tool, have been used in this approach.

Although this approach is basically heuristic, it requires human judgment and decision making; nonetheless, for consistency, the steps and rules have been
adequately defined. The biggest challenge, especially when dealing with a large number of requirements, arises when we fail to find a suitable composition point in the current system architecture for composing it with the partial architecture for the current requirement. The current strategy is to defer the composition of the current partial architecture and compose it with the (current) system architecture when it becomes possible. In the examples, this strategy has been experimented with and proven to work.

On the other hand, we may also need to deal with non-determinism when there is more than one possible composition point. This is bound to arise when dealing with a large number of requirements. There cannot be any hard and fast rule here and human guidance is the only practical solution.

In the analysis of NLR, many other elements could be identified than at present. To date, we have only focused on computation and control. The X-MAN component model can be refined to incorporate elements, e.g. active components, data flow, etc., and indeed different versions of the model are being constructed to accommodate them.

10.3 Future Work

It is recommended that further research be undertaken in the following areas: (1) use of NLP; (2) execution of large-scale set of requirements; (3) execution of a large-scale set of requirements; (4) experimentation with regards to the potential cost and benefit of the approach; (5) architecture refactoring; (6) automation of refactoring; and (7) an integrated tool support.
10.3.1 The Use of Natural Language Processing in the Elements Extraction

This work has adopted a basic textual analysis technique as a means of assisting designers in suggesting candidates for elements of component-based systems. To achieve a higher degree of effectiveness, the use of advanced feature of Natural Language Processing (NLP) technique can further be investigated and exploited in the future. The application of any suitable and advanced technique in NLP can be embedded in the Extractor tool. Nevertheless, we do not suggest that automatic extraction should be applied to completely replace human tasks. It is believed that human justification is still demanded during the selection of the relevant component-based elements.

10.3.2 Execution of a Large-scale Set of Requirements

The current state of the approach is defined based on the execution of many examples and case studies. Considerably more work will need to be done to determine the applicability of the approach in various types of systems. It would be interesting if a large-scale set of requirements can be acquired and experimented with. Due to resource constraints (time and availability), large-scale set of requirements cannot be obtained and executed.

10.3.3 An Experimentation of the Cost and Benefit of the Approach

Further research might investigate and experiment the potential cost and benefits of the approach. This kind of experimentation requires empirical investigation
involving real user of the approach, these being the analysts and designers. Moreover, it would also be interesting to compare experiences of individuals using the approach, and what can be learned from their real-user experiences. Such empirical evidence would assist us in establishing a greater degree of accuracy on the applicability and effectiveness of this approach.

10.3.4 Architecture Refactoring

More work will need to be done to establish architecture refactoring tasks. The present strategy may not work for generic software systems, since it is predicated on the premise that all the key concepts derived from the requirements can be encapsulated. The latter will only be true for systems with restricted behaviour, or for highly compositional domains. In order to overcome this shortcoming, we need to be able to refactor an architecture, specifically the connector hierarchy, such that the behaviour demanded by a new requirement can be correctly added to the architecture. The current design decisions for architecture refactoring are more concerned with achieving simplification of the design, rather than restructuring the design for compositional purposes. An investigation of this latter aim is beneficial in order to allow further composition.

10.3.5 Automation of the Architecture Refactoring

At the current stage of the approach, refactoring tasks are manually identified and performed. It would be interesting to automatically detect the desired patterns of structure that can be refactored. As this refactoring is not the main contribution to this work, this is potential future work to be addressed. Having such an automated process would definitely reduce the designer’s effort and hence, enhance the full potential of the intended refactoring tasks.


10.3.6 An Integrated Tool Support

For the time being, the two tools, namely the Extractor and Exogeneous Composition Framework (ECF) tools used in this approach are independently developed and deployed. An integrated tool support could lead to at least three benefits. Firstly, an increment for each requirement could be easier to manage. At the current state of this research, the extraction process is performed separately and the result of the extraction will be used to model using the ECF tool.

Secondly, the feature to manage component selection in the repository could assist the designer. At the present stage of the work, we assume that the required components are already available in the repository, hence no solution is provided to solve the problems of matching the required components with the existing components in the repository. For brevity, we assume the need to apply the signature matching mechanism for component selection. Thirdly, it should also provide support for recording and managing partial architectures that are deferred, as well as matching them with possible composition points in the current system architecture. Currently, these tasks are managed manually, using the ECF tool.

10.4 Summary

This chapter has presented the contributions to the knowledge made in this research. All the respective parts of the thesis link to each of the contributions have also been provided. In addition, limitations are discussed and finally, the future research directions have been addressed.
Bibliography


[Ach97] C. B Achour. Linguistic instruments for the integration of scenarios


[Brib] Eric Brill. Transformation-based error-driven learning and natural Language P.


Notes in Computer Science, pages 377–392. Springer Berlin / Heidelberg.


[FE00] Anthony Finkelstein and Wolfgang Emmerich. The future of requirements management tools. Information Systems in Public


[Lin07] Ling Ling. *Composing Software Components in Design Phase Using*


Appendix A

Textual Analysis

This appendix sets a brief introduction of the textual analysis technique and the part-of-speech (POS) tagging. The knowledge from both areas is applied and adopted in the element extraction process. With the help of the textual analysis technique, an analyst can concentrate on the selection of the extracted keywords, instead of direct and manual screening from each requirement.

A.1 Introduction

Some background on textual analysis is presented. Linguistic techniques in natural language processing can be categorised into (1) lexical (2) syntactic (3) semantic [Ach97], or (4) pragmatic levels [BMA08].

Lexical approaches, such as implemented in AbstFinder [GB97] identify repetition of words or clauses using statistical analysis techniques in order to find abstractions in requirements. A few other examples that fall into the lexical approaches are Abbott’s textual analysis [Abb83], Saeki et al. [SHE89], Chen [Che80], and Hartman and Link [HL07]. Based on the analysis, these approaches extract the relevant information that can be mapped into domain models, specification or program codes. The Abbott’s approach [Abb83] uses natural language to identify data types, objects and operators. In his approach, he uses common nouns to indicate data types that are classes; proper nouns and direct references to indicate objects such as verbs, attributes, predicates; and descriptive expressions to suggest operators that perform computations; and conditional clauses such as if, then, else, for, do, while etc. to suggest control structures.

The syntactic approaches, on the other hand, analyse a larger chunks of a sentence than individual words [BMA08]. A syntactic parsing method produces labels and the hierarchical structure of a sentence.

Semantic approaches deal with how to represent the meaning of a statement,
linguistic inferences, and word-sense disambiguation (WSD). In the requirements context, we believe it is almost impossible to automatically derive semantic extraction from requirements statements. Although work (e.g. Automatic Programming) has attempted to provide such a link, a significant amount of human effort and intervention is still required.

NLP in the RE context differs from the general purpose NLP in the sense that they require different sets of input and output [BMA08]. Refer to Figure 2 and 3 [BMA08].

Table A.1: Levels of NLP [BMA08]

<table>
<thead>
<tr>
<th>Level of approaches</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>Assign each word using POS tagger</td>
<td>POS tagger (e.g. noun, verb, conjunction, etc.) Corpora: WSJ, Brown Corpus</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Analyse larger chunks of a sentence and represent the structure of a sentence</td>
<td>Probabilistic parsers (e.g. noun phrases, verb phrases, prepositional phrases, etc.) Corpora: WSJ, Brown Corpus</td>
</tr>
<tr>
<td>Semantic</td>
<td>Represent the meaning of a sentence, linguistic inferences and word-sense disambiguation (WSD)</td>
<td>Semantic parsers, WSD classifiers Corpora: FrameNet, Senseval</td>
</tr>
<tr>
<td>Pragmatic</td>
<td>Understand relationships between language and context. (e.g. anaphora resolution to distinguish what pronouns or noun phrases refer to in statements)</td>
<td>Discourse Analysers Corpora: Penn Discourse Treebank</td>
</tr>
</tbody>
</table>

A.2 Part-of-Speech Tagger

This section introduces the POS tagging and provide an example of the POS tagging process.

A.2.1 What is a POS Tagger?

In general, tagging is the process of assigning parts of speech to each word in a sentence. Tagging is an automatic descriptor or tag for a specified input [Mit04]. It is one basic kind of linguistic structure: syntactic word classes which is also referred to as grammatical tagging. Given a requirement statement:

```
The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).
```
The outcome of the POS tagging for this particular requirement is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<output>
<s i="0">
 <token pos="at">The</token>
 <token pos="nps">ATM</token>
 <token pos="md">will</token>
 <token pos="vb">service</token>
 <token pos="cd">one</token>
 <token pos="nn">customer</token>
 <token pos="in">at</token>
 <token pos="at">a</token>
 <token pos="nn">time</token>
 <token pos=".">,<token>
 </s>
<s i="1">
 <token pos="at">A</token>
 <token pos="nn">customer</token>
 <token pos="md">will</token>
 <token pos="be">be</token>
 <token pos="vbn">required</token>
 <token pos="to">to</token>
 <token pos="vb">insert</token>
 <token pos="at">an</token>
 <token pos="nn">ATM</token>
 <token pos="nn">card</token>
 <token pos="cc">and</token>
 <token pos="vb">enter</token>
 <token pos="at">a</token>
 <token pos="jj">personal</token>
 <token pos="nn">identification</token>
 <token pos="nn">number</token>
 <token pos="."),<token>
 </s>
</output>
```

Each word is tagged (originally labelled by an underscore ‘_’ with a tag) with a pair of XML tags. For example ‘The’ is tagged as ‘at’ that is an article; ‘ATM’ as ‘nps’ that is a proper singular or plural noun; ‘will’ as ‘md’ that means modal auxiliary; ‘service’ as ‘vb’ that means verb; etc.

POS tagging can be used as pre-processing of raw texts, information retrieval, linguistic studies, speech processing, and in IT applications [Mit04].

The general basic POS tagging processes [Mit04] are *tokenisation, ambiguity*
look-up and ambiguity resolution. During tokenisation, each word in the statement is tokenised for analysis purposes. In ambiguity look-up, a compiler or interpreter assigns all the possible tags for each word accordingly. A guesser might also be used when any of the words are not tagged according to the pre-defined tag set. Finally, to resolve ambiguity, based on (1) information about the word itself, e.g. some words are typically nouns rather than verbs such as love, angle (2) information about the tag sequences, e.g. if a preceding word is a preposition, the POS tagging model might prioritise the noun analyser rather than the verb analyser.

POS tagging can be either supervised (that is a pre-defined corpora is required in order to train the tag set) or unsupervised (does not need a pre-defined corpora, but uses an algorithm instead). There are two POS tagging methods: (1) Rule-based Tagging or Transformation-based Tagging, for example ENGT-WOL [Vou95], Brill's tagger [Bria, Brib], (and (2) Statistical (Stochastic) Tagging, for example Trigrams ‘N’ Tags(TNT) [Bra00].

In general, rule-based tagging defines a dictionary and each word will be tagged based on the defined dictionary. In order to remove certain tags, rules are manually created. Whilst in statistical tagging, probability theories are adopted, for instance the “most-frequent-tag” algorithm. One of the best known corporuses that is a collection of linguistically annotated text, is the Brown University Standard Corpus of Present-Day American English (known as the Brown Corpus).\\(^1\) The corpus contains approximately 1,014,312 words.

The general POS categories are noun, verb, article, preposition, pronoun, adverb, conjunction and interjection. However, the issue with POS tagger is ambiguity. In reality, a single word may have multiple parts of speech which need to be resolved.

\(^1\)The Brown Corpus Manual can be accessed at http://icame.uib.no/brown/bcm.html
Appendix B

Questionnaire
Dear participants,

Thank you for your effort to participate in this experiment. In brief, this experiment is designed to validate the proposed incremental approach to constructing component-based systems from natural language requirements (NLR) using a specific component model i.e. Exogeneous Component Model (XMAN).

Prior to the experiment, you will be trained on:

1. Fundamentals of the XMAN semantics, element extraction and mapping processes;
2. Using the Extractor tool;

Kindly take note that during the experiment, your analysis and design decisions should not be biased on the provided tools. The expected outcome from your tasks is the system design for the given set of requirements using XMAN model. This design should be derived incrementally from the NLR.

This set of questionnaire consists of:

1. Set A: Demographic survey
2. Set B: During experiment questionnaire items
3. Set C: Post experiment questionnaire items

You are required to fill in all of the above items. All responses will be treated in the strictest confidence.

Thank you.
SET A: DEMOGRAPHIC SURVEY

ID: ____________________

1. Gender (please tick):
   - [ ] Male
   - [ ] Female

2. Ethnicity/Race (please tick):
   - [ ] Malay
   - [ ] Chinese
   - [ ] Indian
   - [ ] Bumiputera-non Malay
   - [ ] Others (please specify): ____________________

3. Indicate the number of years of your working experience related to IT industry (please tick)*:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1-3</th>
<th>4-6</th>
<th>More than 7</th>
<th>Additional Note</th>
</tr>
</thead>
</table>

*If you are a student, please tick here [ ] and indicate the current year in the above table.

4. Your highest degree level (please tick):
   - [ ] PhD
   - [ ] Masters
   - [ ] Diploma
   - [ ] Bachelor Degree
   - [ ] Others (please specify): ____________________

5. On a scale from 1 - 5, how do you rate your software modelling competency level? (Please tick ONE)
   - [ ] 1. Very Poor
   - [ ] 2. Poor
   - [ ] 3. Fair
   - [ ] 4. Good
   - [ ] 5. Outstanding

6. Have you been involved in software development? (Please tick) If YES, state the number of years:
   - [ ] Yes, ________________  [ ] No

7. Are you familiar with any of the following software development methodology?
   - [ ] Waterfall
   - [ ] Agile
   - [ ] Rapid Prototyping/JAD
   - [ ] Unified Process
   - [ ] Others (Please specify): ____________________
   - [ ] None

8. Have you had any experience in developing component OR software system for reuse (i.e. Component-based software development)? (Please tick):
   - [ ] Yes
   - [ ] No

9. Have you had any experience in using OR modelling component-based software engineering? (Please tick):
   - [ ] Yes
   - [ ] No
EXPERIMENT ON EXTRACTION & MAPPING OF ELEMENTS FROM NATURAL LANGUAGE REQUIREMENTS (NLR)

Brief Description
You are required to execute the Extractor tool and extract candidates of component-based elements, i.e., computation, conceptual component, control and data (whenever required) from NLR statements. During the extraction process, the main aim is to identify which of the heuristics is relevant to be adopted. In the second task, you are required to represent the extracted element by modelling those elements according to the XMAN semantics. Both of these tasks MUST be performed iteratively and incrementally until all the given requirements are completed.

Preparation BEFORE the experiment
The requirements are provided in EIGHT (8) separated files. You will be able to find these files in the ATM folder on the desktop.

TASK 1: EXTRACTION OF NATURAL LANGUAGE REQUIREMENTS

1. First, execute the Extractor tool.
2. Perform the extraction process i.e., TASK 1. Start with browsing the input file for Requirement-1 from the ATM folder. (You may refer to the Extractor tool tutorial).
   a. Answer Q1 of the Questionnaire Item.
3. Proceed with TASK 2.
4. Repeat Steps 2-3 for a new requirement statement.
5. Press <Save> button once you completed the syntax. The newly created composition will appear at the bottom list. (Tips: You may use this in the other composition steps.)
6. Click <Complete> button. You will be prompted with the previous interface. Click <View Result> button to display the result.

TASK 2: MODELLING OF THE EXTRACTION ELEMENTS

1. For each iteration of a particular requirement, use the result from extraction process to model the extracted elements. (Refer to The Exogeneous Composition Framework (ECF) Tool Tutorial).
2. Once completed, if you are dealing with the final requirement, proceed with Steps 5-6. Otherwise, if this is not the final requirement, continue the iteration by repeating Steps 2-4.
QUESTIONNAIRE ITEMS

Q1. Requirement-1: The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN).

TASK 1: ELEMENT EXTRACTION (Refer Appendix 1)

1. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   a. Identifying computations
   b. Identifying computations from user interaction
   c. Identifying conceptual component
   d. Identifying control
   e. Identifying data
   f. NONE, explain (____________________________)

TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS (Refer Appendix 2)

2. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?
   a. Map control flows to composition connectors
   b. Choose between PIPE or SEQ
   c. Map control to a GUARD
   d. Combine or separate connectors
   e. Combine or separate components
   f. Computation inferences
   g. Handling data elements
   h. No indication of any computation or control
   i. NONE, explain (_______________________)

Q2. Requirement-2: A customer must be able to make cash withdrawal from the linked account. Approval must be obtained from the bank before cash is dispensed.

TASK 1: ELEMENT EXTRACTION (Refer Appendix 1)

3. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   a. Identifying computations
   b. Identifying computations from user interaction
   c. Identifying conceptual component
   d. Identifying control
   e. Identifying data
   f. NONE, explain (____________________________)
TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS
(Refer Appendix 2)

4. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?
   a. Map control flows to composition connectors  
   b. Choose between PIPE or SEQ  
   c. Map control to a GUARD  
   d. Combine or separate connectors  
   e. Combine or separate components  
   f. Computation inferences  
   g. Handling data elements  
   h. No indication of any computation or control  
   i. NONE, explain (_______________________)

Q3. Requirement-3: A customer must be able to deposit cash to the linked account that can be inserted to the cash slot. Approval must be obtained from the bank before physically accepting the cash.

TASK 1: ELEMENT EXTRACTION (Refer Appendix 1)

5. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   a. Identifying computations  
   b. Identifying computations from user interaction  
   c. Identifying conceptual component  
   d. Identifying control  
   e. Identifying data  
   f. NONE, explain (____________________________)

TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS
(Refer Appendix 2)

6. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?
   a. Map control flows to composition connectors  
   b. Choose between PIPE or SEQ  
   c. Map control to a GUARD  
   d. Combine or separate connectors  
   e. Combine or separate components  
   f. Computation inferences  
   g. Handling data elements  
   h. No indication of any computation or control  
   i. NONE, explain (____________________________)
Q4. Requirement-4: A customer must be able to make a transfer of money between any two accounts originated from the linked account.

**TASK 1: ELEMENT EXTRACTION** (Refer Appendix 1)

7. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   - Identifying computations
   - Identifying computations from user interaction
   - Identifying conceptual component
   - Identifying control
   - Identifying data
   - NONE, explain (____________________________)

**TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS** (Refer Appendix 2)

8. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?
   - Map control flows to composition connectors
   - Choose between PIPE or SEQ
   - Map control to a GUARD
   - Combine or separate connectors
   - Combine or separate components
   - Computation inferences
   - Handling data elements
   - No indication of any computation or control
   - NONE, explain (____________________________)

Q5. Requirement-5: A customer must be able to make a balance inquiry of the linked account.

**TASK 1: ELEMENT EXTRACTION** (Refer Appendix 1)

9. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   - Identifying computations
   - Identifying computations from user interaction
   - Identifying conceptual component
   - Identifying control
   - Identifying data
   - NONE, explain (____________________________)
Q6. Requirement-6: If the customer fails to be authenticated, the card will be rejected.

TASK 1: ELEMENT EXTRACTION (Refer Appendix 1)

11. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?
   a. Identifying computations
   b. Identifying computations from user interaction
   c. Identifying conceptual component
   d. Identifying control
   e. Identifying data
   f. NONE, explain (____________________________)
Q7. Requirement-7: After each transaction, the ATM will display and print a receipt containing the transaction information.

**TASK 1: ELEMENT EXTRACTION** (Refer Appendix 1)

13. Which heuristic(s) that you applied when you made your decision on extracting the relevant element for the above requirement?

   a. Identifying computations
   b. Identifying computations from user interaction
   c. Identifying conceptual component
   d. Identifying control
   e. Identifying data
   f. NONE, explain (____________________________)

**TASK 2: MAPPING OF THE EXTRACTED ELEMENT TO COMPONENT-BASED ELEMENTS** (Refer Appendix 2)

14. Which mapping heuristic(s) that you applied when you made your decision on mapping the extracted element for the above requirement?

   a. Map control flows to composition connectors
   b. Choose between PIPE or SEQ
   c. Map control to a GUARD
   d. Combine or separate connectors
   e. Combine or separate components
   f. Computation inferences
   g. Handling data elements
   h. No indication of any computation or control
   i. NONE, explain (____________________________)
APPENDIX 1: Summary of Element Extraction Heuristics

<table>
<thead>
<tr>
<th>Summary of Element Extraction Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[HR-1A] Identifying computations</strong></td>
</tr>
<tr>
<td>Computation can be identified using textual analysis based on (1) category of verbs, (explicit) data transformation, (implicit) state and event, (2) action nouns and (3) phrases, descriptive descriptions and predicate.</td>
</tr>
<tr>
<td><strong>[HR-1B] Identifying computations from user interaction</strong></td>
</tr>
<tr>
<td>Computation can be implied from any interaction between users and systems or hardware devices.</td>
</tr>
<tr>
<td><strong>[HR-2] Identifying conceptual component</strong></td>
</tr>
<tr>
<td>Conceptual components can be identified directly from action nouns (based on textual analysis), or implied from hardware devices or a system.</td>
</tr>
<tr>
<td><strong>[HR-3] Identifying control</strong></td>
</tr>
<tr>
<td>Control can be (1) explicitly identified using control structure in textual analysis i.e., from conjunctions or prepositions (2) identified using specific terms that semantically represent control flow (3) implied from preposition in text.</td>
</tr>
<tr>
<td><strong>[HR-4] Identifying data</strong></td>
</tr>
<tr>
<td>Data can be identified and implied from the extracted nouns.</td>
</tr>
</tbody>
</table>
APPENDIX 2: Summary of Mapping Heuristics

<table>
<thead>
<tr>
<th>Summary of Mapping Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[HMR-1]</strong> Design decision to map control flows to composition connectors</td>
</tr>
<tr>
<td>Decision to map control flow to a composition connector is based on either ordering or branching (selection) execution.</td>
</tr>
<tr>
<td><strong>[HMR-2]</strong> Design decision to choose between a PIPE or a SEQ composition connector</td>
</tr>
<tr>
<td>Decision to choose between a PIPE and a SEQ is based on the availability of data (what data is required in the connecting component)</td>
</tr>
<tr>
<td><strong>[HMR-3]</strong> Design decision to map control to a GUARD</td>
</tr>
<tr>
<td>Guards can be identified when (1) there is a constraint on a single component; (2) there is data to be passed to the connecting component.</td>
</tr>
<tr>
<td><strong>[HMR-4A]</strong> Deciding whether to combine or separate connectors</td>
</tr>
<tr>
<td>When deciding the number of connector(s) when dealing with more than two components in a single design, if we anticipate the composition has a high probability of being reused, use a separate connector.</td>
</tr>
<tr>
<td><strong>[HMR-4B]</strong> Deciding whether to combine computations into a single component or separate computations into different components</td>
</tr>
<tr>
<td>Group computations that are provided by the same component (especially for hardware devices); the rest of the computations can be modeled separately.</td>
</tr>
<tr>
<td><strong>[HMR-5]</strong> Computation inferences</td>
</tr>
<tr>
<td>When a requirement does not explicitly indicate a specific computation but obviously we need to provide a computation, we can imply the computation from phrases in the text.</td>
</tr>
<tr>
<td><strong>[HMR-6]</strong> Handling data elements</td>
</tr>
<tr>
<td>When a requirement contains only information on data elements, document them. Data can be used as constraints in branching, store values, or initialise values.</td>
</tr>
<tr>
<td><strong>[HMR-7]</strong> No indication of any computation or control</td>
</tr>
<tr>
<td>Assume the order in which the requirement is written.</td>
</tr>
</tbody>
</table>
SET C: POST EXPERIMENT QUESTIONNAIRE ITEMS

1. During the training, evaluate the difficulty levels for each of these items in order to understand the provided materials. (Tick ONE for each item)

<table>
<thead>
<tr>
<th>Training material</th>
<th>Very Easy</th>
<th>Easy</th>
<th>Average</th>
<th>Difficult</th>
<th>Very Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. XMAN fundamentals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. Element Extraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. The Extractor Tool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d. Mapping to component-based elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e. The ECF Tool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. When using the approach, rate 1-6 (1-very easy, 6-very difficult) on the tasks that require the most mental effort (Assume that this exclude the learning time acquired during the training session.)

<table>
<thead>
<tr>
<th>Task</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Extracting candidates of component-based system.</td>
<td></td>
</tr>
<tr>
<td>b. Identifying which extraction heuristics is relevant for a specific requirement.</td>
<td></td>
</tr>
<tr>
<td>c. Analysing the interaction (behaviour).</td>
<td></td>
</tr>
<tr>
<td>d. Modelling the composition using XMAN.</td>
<td></td>
</tr>
<tr>
<td>e. Identifying which mapping heuristic(s) is to be applied for a particular extraction.</td>
<td></td>
</tr>
<tr>
<td>f. Handling incremental composition.</td>
<td></td>
</tr>
</tbody>
</table>

3. How do you think the incremental approach can be improved?

_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________

-THE END-
Appendix C

Case Studies
This appendix contains the complete documentation of the execution of the case studies and example. The first case study is the Trading System (COCOME), which is used for handling sales transactions in a supermarket. This set of requirements has also been used as a reference example for component-based development [RRMP08]. The second case study is that of the Video Store System (VSS), which is taken from [LSB98]. The third example is the Word Count, which is drawn from [Sch06, p. 434].

C.1 The Trading System (COCOME)

C.1.1 COCOME Requirements

The Trading System is used for handling sales transactions in a supermarket. This set of requirements has also been used as a reference example for component-based development [RRMP08]. The system comprises nine main functions including (1) Process Sale, which handles Cash Desk operations; (2) Manage Express Checkout, which deals with transaction modes, i.e. normal and express; (3) Order Products, which allows the Store Manager to order products from suppliers; (4) Receive Ordered Products, which allows the Store Manager to update received orders; (5) Show Stock Reports, which permits the Store Manager to view all available stock; (6) Show Delivery Reports, which allows the Enterprise Manager to generate reports; (7) Change Price, which permits the Store Manager to update a product’s price; (8) Product Exchange (on low stock) Among Stores, which handles product orders between stores and (9) Remove Incoming Status, which allows the Store Manager to update the received product. Altogether, there are 47 requirements.

C.1.2 UC 1 – Process Sale

*Brief Description:* At the Cash Desk the products a Customer wants to buy are detected and the payment – either by credit card or cash – is performed.

*Involved Actors:* Customer, Cashier, Bank, Printer, Card Reader, Cash Box, Bar Code Scanner, Light, Display.

*Precondition:* The Cash Desk and the Cashier are ready to start a new sale.

*Trigger:* Coming to the Cash Desk a Customer wants to pay for his chosen product items.

*Postcondition:* The Customer has paid, has received the bill and the sale is registered in the Inventory.

*Standard Process*

1. The Customer arrives at the Cash Desk with goods to purchase.
2. The Cashier starts a new sale by pressing the button Start New Sale at the Cash Box.

3. The Cashier enters the item identifier. This can be done manually by using the keyboard of the Cash Box or by using the Bar Code Scanner.

4. Using the item identifier the System presents the corresponding product description, price, and running total. Steps 3-4 are repeated until all items are registered.

5. Denoting the end of entering items the Cashier presses the button Sale Finished at the Cash Box.

(a) To initiate cash payment the Cashier presses the button Cash Payment at the Cash Box.

   i. The Customer hands over the money for payment.
   ii. The Cashier enters the received cash using the Cash Box and confirms this by pressing Enter.
   iii. The Cash Box opens.
   iv. The received money and the change amount are displayed, and the Cashier hands over the change.
   v. The Cashier closes the Cash Box.

(b) In order to initiate card payment the Cashier presses the button Card Payment at the Cash Box.

   i. The Cashier receives the credit card from the Customer and pulls it through the Card Reader.
   ii. The Customer enters his PIN using the keyboard of the card reader and waits for validation.
   iii. Step 5.b.ii is repeated until a successful validation or the Cashier presses the button for cash payment.

6. Completed sales are logged by the Trading System and sale information is sent to the Inventory in order to update the stock.

7. The Printer writes the receipt and the Cashier hands it out to the Customer.

8. The Customer leaves the Cash Desk with receipt and goods.

Alternative or Exceptional Processes

1. In step 3: Invalid item identifier if the system cannot find it in the Inventory.

   (a) The System signals an error and rejects this entry.
   (b) The Cashier can respond to the error as follows:
i. A human-readable item identifier exists:
   A. The Cashier manually enters the item identifier.
   B. The System displays the description and price.
ii. Otherwise, the product item is rejected.

2. In step 5.b: Card validation fails.
   (a) The Cashier and the Customer try again and again.
   (b) Otherwise, the Cashier requires the Customer to pay cash.

3. In step 6: Inventory not available.
   (a) The System caches each sale and writes them into the Inventory as soon as it is available again.

C.1.3 UC 2 – Manage Express Checkout

*Brief Description:* If some conditions are fulfilled a Cash Desk automatically switches into express mode. The Cashier is able to switch back into normal mode by pressing a button at his Cash Desk. To indicate the mode the Light Display shows different colours.

*Involved Actors:* Cashier, Cash Box, Light Display, Card Reader.

*Precondition:* The Cash Desk is either in normal mode and the latest sale was finished (case 1) or the Cash Desk is in express mode (case 2).

*Trigger:* This use case is triggered by the system itself.

*Postcondition:* The Cash Desk has been switched into express mode or normal mode. The Light Display has changed its colour accordingly.

*Standard Process*

1. The considered Cash Desk is in normal mode and has just finished a sale which matches the condition of an express checkout sale. Now 50% of all sales during the last 60 minutes fulfill the condition for an express checkout.
   (a) This Cash Desk, which has caused the achievement of the condition, is switched into express mode.
   (b) Furthermore the corresponding Light Display is switched from black into green to indicate the Cash Desk’s express mode.
   (c) Paying by credit card is not possible any more.
   (d) The maximum of items per sale is reduced to 8 and only paying by cash is allowed.

2. The Cash Desk is in express mode and the Cashier decides to change back into normal mode.
(a) The Cashier presses the button Disable Express Mode.
(b) The colour of the Light Display is changed from green to black colour.

3. Cash and also card payment is allowed and the Customer is allowed to buy as many goods as he likes.

C.1.4 UC 3 – Order Products

*Brief Description:* The Trading System provide the opportunity to order product items.

*Involved Actors:* Store Manager.

*Precondition:* An Overview over the Inventory is available and the Store Client was started.

*Trigger:* The Store Manager decided to buy new product items for his store.

*Postcondition:* The order was placed and a generated order identifier was presented to the Store Manager.

**Standard Process**

1. A list with all products and a list with products running out of stock are shown.
2. The Store Manager chooses the product items to order and enters the corresponding amount.
3. The Store Manager presses the button Order at the Store Client’s GUI.
4. The appropriate suppliers are chosen and orders for each supplier are placed. An order identifier is generated for each order and is shown to the Store Manager.

C.1.5 UC 4 – Receive Ordered Products

*Brief Description:* Ordered products which arrive at the Store have to be checked for correctness and inventoried.

*Involved Actors:* Stock Manager.

*Precondition:* The Store Client was started and the part Inventory of the Trading System is available.

*Trigger:* The ordered products arrive at the Store.

*Postcondition:* The Inventory is updated with the ordered products.

**Standard Process**

1. Ordered products arrive at the stock attached by an order identifier which has been assigned during the ordering process.
2. The Stock Manager checks the delivery for completeness and correctness.

3. In the case of correctness, the Stock Manager enters the order identifier and presses the button Roll in received order.

4. The Trading System updates the Inventory.

Alternative or Exceptional Processes

1. In step 2: Delivery not complete or not correct.
   The products are sent back to the supplier and the Stock Manager has to wait until a correct and complete delivery has arrived. This action is not recognised by the System.

C.1.6 UC 5 – Show Stock Reports

Brief Description: The opportunity to generate stock-related reports is provided by the Trading System.
Involved Actors: Store Manager.
Precondition: The reporting GUI at the Store Client has been started.
Trigger: The Store Manager wants to see statistics about his store.
Postcondition: The report for the Store has been generated and is displayed on the reporting GUI.

Standard Process

1. The Store Manager enters the store identifier and presses the button Create Report.

2. A report including all available stock items in the store is displayed.

C.1.7 UC 6 – Show Delivery Reports

Brief Description: The Trading System provides the opportunity to calculate the mean times a delivery from each supplier to an considered enterprise takes.
Involved Actors: Enterprise Manager.
Precondition: The reporting GUI at the Store Client has been started.
Trigger: The Enterprise Manager wants to see statistics about the enterprise.
Postcondition: The report for the Enterprise has been generated and is displayed to the Enterprise Manager.

Standard Process

1. The Enterprise Manager enters the enterprise identifier and presses the button Create Report.
2. A report which informs about the mean times is generated.

C.1.8 UC 7 – Change Price

Brief Description: The System provides the opportunity to change the sales price for a product.
Involved Actors: Store Manager.
Precondition: The store GUI at the Store Client has been started.
Trigger: The Store Manager wants to change the sales price of a product for his store.
Postcondition: The price for the considered product has been changed and it will now be sold with the new price.

Standard Process

1. The System presents an overview over all available products in the store.
2. The Store Manager selects a product item and changes its sales price.
3. The Store Manager commits the change by pressing ENTER.

C.1.9 UC 8 – Product Exchange (on low stock) Among Stores

Brief Description: If a store runs out of a certain product (or a set of products; “required good”), it is possible to start a query to check whether those products are available at other Stores of the Enterprise (“providing Stores”). Therefore the Enterprise Server and the Store Servers need to synchronize their data on demand (one scheduled update per day or per hour is not sufficient). After a successful query the critical product can be shipped from one to other Stores. But it has to be decided (using heuristics to compute the future selling frequency), whether the transportation is meaningful. For example, if the product is probably sold out at all Stores within the same day, a transportation does not make sense.

Expressed in a more technical way, a Store Server is able to start a query at the Enterprise Server. The Enterprise Server in turn starts a query for products available at other Stores. As the Enterprise Server does not have the current global data for Stores at any time (due to a write caching latency at the Store Servers) the Enterprise Server has to trigger all Store Servers to push their local data to the Enterprise Server.
Involved Actors: This use case is not an end-user use case. Only servers are involved.
Precondition: The Store Server with the shortage product is able to connect to the Enterprise Server.
Trigger: This use case is triggered by the system itself.
Postcondition: The products to deliver are marked as incoming or unavailable, respectively, in the according Stores.

**Standard Process**

1. A certain product of the Store runs out.
2. The Store Server recognises low stock of the product.
3. The Store Server sends a request to the Enterprise Server (including an identification of the shortage products, and a Store id).
4. The Enterprise Server triggers all Stores that are "nearby" (e.g. 300 km) the requiring store, to flush their local write caches. So the Enterprise Server database gets updated by the Store Server.
5. The Enterprise Server does a database look-up for the required products to get a list of products (including amounts) that are available at providing Stores.
6. The Enterprise Server applies the "optimization criterion" (specified above) to decide whether it is meaningful to transport the shortage product from one store to another (heuristics might be applied to minimize the total costs of transportation). This results in a list of products (including amounts) per providing store that has to be delivered to the requiring Store.
7. The Store Server, initially sending the recognition of the shortage product, is provided with the decision of the Enterprise Server.
   (a) The required product is marked as incoming.
8. The Store Server of a nearby Store is provided with information that it has to deliver the product.
   (a) The required product is marked as unavailable in the Store.

**Alternative or Exceptional Processes**

1. The Enterprise Server is not available: The request is queued until the Enterprise Server is available and then is sent again.
2. One or more Store Servers are not available: The Enterprise Server queues the requests for the Store Servers until they are available and then resends them.
3. If a Store Server is not available for more than 15 minutes the request for this Server is cancelled. It is assumed that finally unavailable Store Servers do not have the required product.
C.1.10 Extension on UC 8 – Remove Incoming Status

Brief Description: If the first part of use case 8 (as described above) has passed, for moved products an amount marked as incoming remains at the Inventory of the Store receiving the products. An extension allows to change that incoming mark via a user interface at the Store Client if the moved products arrive at a Store.

Precondition: The Inventory is available and the Store Client has been started.

Trigger: The moved products (according to UC8) arrive at the Store.

Postcondition: For the amount of incoming products the status “incoming” is removed in the Inventory.

Standard Process

1. The products arrive at the stock of the Store.
2. For all arriving products the Stock Manager counts the incoming amount.
3. For every arriving product the Stock Manager enters the identifier and its amount into the Store Client.
4. The system updates the Inventory.

Alternative or Exceptional Processes

1. If the entered amount of an incoming product is larger than the amount accounted in the Inventory, the input is rejected. The incoming amount has to be re-entered.

C.1.11 Incremental Execution of the COCOME Requirements

This section explains how each requirement in natural language is handled in each of the steps defined in the approach. To simplify the documentation, each step will be labelled as Step-1 to Step-5 accordingly. To recall:

Step-1 Extracting elements of component-based systems from requirements.
Step-2 Mapping of the extracted elements to X-MAN elements.
Step-3 Creating a partial architecture.
Step-4 Composing the partial architecture with the existing system architecture.
Step-5 Finalising the system architecture.

Note that Step-5 is only applicable after the final requirement has been executed. For each increment, whenever applicable, any issue and discussion is also added.
C.1.12 Use Case 1 – Process Sale

Requirement-UC1-1

1. The Customer arrives at the Cash Desk with goods to purchase.

**Step-1** This is not a functional requirement. Hence, all the following steps for this increment are skipped.

**Step-2** Not applicable.

**Step-3** Not applicable.

**Step-4** Not applicable.

**Issue and discussion** Not applicable.

Requirement-UC1-2

2. The Cashier starts a new sale by pressing the button Start New Sale at the Cash Box.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. starts new</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.1.

**Step-3** Since only a single component is identified, no partial architecture is modelled.

**Step-4** Not applicable.

**Issue and discussion** Not applicable.
Table C.2: Summary of Steps 1 & 2 for UC1-3

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. enters item id</td>
<td>DT readItemId → CashBox (CB)</td>
<td></td>
<td>HM-1B</td>
<td></td>
</tr>
<tr>
<td>Comp. enters item id</td>
<td>DT readItemId → Bar Code System (BCS)</td>
<td></td>
<td>HM-1B</td>
<td></td>
</tr>
<tr>
<td>Control ‘or’ conjunction</td>
<td>SEL</td>
<td></td>
<td>HM-2</td>
<td>SEL(readItemId → CB, readItemId → BCS)</td>
</tr>
<tr>
<td>Data choice</td>
<td>Implicit data</td>
<td>Store the selection</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

Requirement-UC1-3

3. The Cashier enters the item identifier. This can be done manually by using the keyboard of the Cash Box or by using the Bar Code Scanner.

Step-1 & Step-2 See Table C.2.

Step-3 Here, a SEL can directly be applied to both the components that provide the computations to be selected. The partial architecture for this requirement is illustrated in Fig. C.1.

![Fig. C.1: UC1-R3](image)

Step-4 As this composition is the first incremental architecture, it is considered as the initial system architecture.

Issue and discussion The assumption here is that this composition works as an Item Identifier (II).
C.1. THE TRADING SYSTEM (COCOME)

Requirement-UC1-4

4. Using the item identifier the System presents the corresponding product description, price, and running total. The steps 3-4 are repeated until all items are registered.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>presents</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>item identifier</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>'using' implies ordering PIPE</td>
<td>HM-2</td>
</tr>
<tr>
<td></td>
<td>'repeat' pre-defined repetition keyword LOOP adapter (HM-3)</td>
<td>LOOP((complete ≠ no) PIPE(identifyItem -II, display -INV))</td>
</tr>
<tr>
<td>Data</td>
<td>productDesc, Data store values for productDesc, price, runningTotal</td>
<td>HM-4</td>
</tr>
<tr>
<td></td>
<td>price, runningTotal</td>
<td></td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.3. In order to display the item information, a search computation is required. We imply such a data transformation for searching. The loop continues until the constraint turns true (the item identification process is completed.)

Step-3 The partial architecture for this requirement is illustrated in Fig. C.2(a).
**Step-4** The composition for UC1-R4 is composed with the current system architecture as shown in Fig. C.2(b)

**Issue and discussion** The composition of CB and BCS components has the same behaviour as II component. Therefore, it can be used to replace the II component because the composition has already been created in the former increment.

**Requirement-UC1-5**

5. Denoting the end of entering items the Cashier presses the button Sale Finished at the Cash Box.

   (a) To initiate cash payment the Cashier presses the button Cash Payment at the Cash Box.

   i. The Customer hands over the money for payment.

   ii. The Cashier enters the received cash using the Cash Box and confirms this by pressing Enter.

   iii. The Cash Box opens.

   iv. The received money and the change amount are displayed, and the Cashier hands over the change.

   v. The Cashier closes the Cash Box.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>DT</td>
<td>open → CB</td>
</tr>
<tr>
<td>change</td>
<td>Descriptive</td>
<td>calcChge → CC</td>
</tr>
<tr>
<td>amount</td>
<td>expression</td>
<td>display → DC</td>
</tr>
<tr>
<td>displayed</td>
<td>DT</td>
<td>close → CB</td>
</tr>
<tr>
<td>closes</td>
<td>DT</td>
<td></td>
</tr>
</tbody>
</table>

**Table C.4: Summary of Steps 1 & 2 for UC1-5**

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>default</td>
<td>PIPE (HM-2)</td>
</tr>
<tr>
<td>assumption</td>
<td>implies ordering</td>
<td></td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.4.
C.1. THE TRADING SYSTEM (COCOME)

Step-3 The partial architecture for this requirement is illustrated in Fig. C.3(a).

![Architecture Diagram](image)

Step-4 The composition for UC1-R5 is composed to the current system architecture as shown in Fig. C.3(b).

**Issue and discussion** The main task of an analyst in identifying elements to be extracted is to differentiate between computations i.e. services provided by components and interactions between users via the interface e.g. GUI or devices. In this case, clearly pressing a button indicates an interaction between user and the system interface. Hence, in this requirement, when the system receives a notification that the entering item process is completed, we need to assign a component that deals with the corresponding data. However, we do not have enough information to assign a specific component other than CB component. The CB component here is not the physical CB component, but can be rather considered as the driver part (or software) of the CB physical component.

Although syntactically, we can directly compose the partial architecture at the front of R5a current architecture, this architecture will not correctly represent the intended behaviour. Another option is to compose using a new PIPE connector via the existing top level connector.

During incremental composition, we may need to add some constraints in order to allow the execution flow to correctly represents the required behaviour. In this case, we add a guard to check that the entering item task is finished and the cash payments selected. The updated architecture is as shown in Fig. C.3(c).

**Requirement-UC1-5b**

(b) In order to initiate card payment the Cashier presses the button Card Payment at the Cash Box.
i. The Cashier receives the credit card from the Customer and pulls it through the Card Reader.

ii. The Customer enters his PIN using the keyboard of the card reader and waits for validation.

iii. The step 5.b.ii is repeated until a successful validation or the Cashier presses the button for cash payment.

Table C.5: Summary of Steps 1 & 2 for UC1-5b

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>Extraction elements</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>read card</td>
<td>DT</td>
<td>readCard → CR</td>
</tr>
<tr>
<td>enters PIN</td>
<td>DT</td>
<td>readPIN → CardReader (CR)</td>
</tr>
<tr>
<td>validation</td>
<td>Action noun</td>
<td>validate → Authentication (AUT)</td>
</tr>
</tbody>
</table>

Control
- ‘repeat’ implies repetition
- ‘or’ implies branching
- ‘and’ ordering

Data card Data store card information

**Step-1 & Step-2** See Table C.5.

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.4(a).

**Step-4** The composition for UC1-R5b is composed to the end of the top-level PIPE connector of the current system architecture as shown in Fig. C.4(b).

**Issue and discussion** Since readPIN is also provided by CR component, we assume it is an internal execution, thus only readCard is exposed as the interface of CR. The architecture after being refactored is shown in Fig. C.4(c).

We need a guard to check the constraint that the user selects this option only when the entering item has completed and the payment is made by card.
C.1. THE TRADING SYSTEM (COCOME)

6. Completed sales are logged by the Trading System and sale information are sent to the Inventory in order to update the stock.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. update log</td>
<td>Comp. up date DT up date</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>Control ‘and’ implies ordering</td>
<td>PIPE (HM-2)</td>
<td>PIPE(log → LOG, update → INV)</td>
</tr>
<tr>
<td>Data saleInfo</td>
<td>Data</td>
<td>store sale info</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.6.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.5(a).

Step-4 For this requirement, the resulting partial architecture is composed to the end of the top-level PIPE as illustrated in Fig. C.5(b).

Issue and discussion Not applicable.

Requirement-UC1-7

7. The Printer writes the receipt and the Cashier hands it out to the Costumer.

Step-1 & Step-2 See Table C.7. Only a single component is identified.

Step-3 Hence, there is no partial architecture for this requirement.
### Table C.7: Summary of Steps 1 & 2 for UC1-7

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. writes</td>
<td>DT</td>
<td>print →PRT</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>conjunction</td>
<td></td>
<td>default assumption</td>
<td>'and' here is irrelevant</td>
</tr>
<tr>
<td>Data receipt</td>
<td>Data transaction info</td>
<td></td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

#### Step-4
The current system architecture is illustrated in Fig. C.6.

#### Issue and discussion
Although there is no control identified here, we assume naturally the receipt contains the list of items bought by the customer. Hence, the flow must be providing some sort of ordering, i.e. after the payment is made, then only the receipt will be printed.
Requirement-UC1-8

8. The Customer leaves the Cash Desk with receipt and goods.

**Step-1** This is not a functional requirement. Hence, all the following steps for this increment are skipped.

**Step-2** Not applicable.

**Step-3** Not applicable.

**Step-4** Not applicable.

**Issue and discussion** Not applicable.

Requirement-UC1-9

9. In step 3: Invalid item identifier if the system cannot find it in the Inventory.

1. The System signals error and rejects this entry.

2. The Cashier can respond to the error as follows:
   
   (a) A human-readable item identifier exists:
       
       i. The Cashier manually enters the item identifier.
       
       ii. The System displays the description and price.

   (b) Otherwise the product item is rejected.

**Step-1 ** & **Step-2** See Table C.8.

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.7(a). Initially, the item information is displayed without any intention of checking for any exception. In this requirement, if the item cannot be found, an error will be displayed. Hence, we add a guard to check that the item is found before the display computation is performed as shown in Fig. C.7(b). Now, looking at the resulting pattern, we can simplify the mutual exclusive guards and replace with a SEL. In addition, the same guards that handle the fault value can be combined as a SEQ (see Fig. C.7(c)).

**Step-4** The composition of UC1-R9 is added to the current system architecture as illustrated in Fig. C.8.

**Issue and discussion** In this requirement, we performed a series of refactorings as discussed in Chapter 7.
<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>identify DT</td>
<td></td>
<td>identify → II</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>search DT</td>
<td></td>
<td>search → INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>display DT</td>
<td></td>
<td>disError → ERR</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>reject DT</td>
<td></td>
<td>reject → REJ</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>‘otherwise’ implies ordering</td>
<td>SEL or guard (HM-3)</td>
<td>PIPE (HM-2)</td>
<td>PIPE(identify → II, search → INV, GUARD(found=N) → display → ERR, GUARD(found=N) &gt; reject → REJ)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data found</th>
<th>Data boolean(N=no)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HM-4</td>
</tr>
</tbody>
</table>

(a)UC1-R9 (b)Refactored architecture (c) Refactored architecture

Fig. C.7: UC1-R9

Fig. C.8: Incremental composition of UC1-R9

**Requirement-UC1-10**

10. In step 5.b: Card validation fails.
Table C.9: Summary of Steps 1 & 2 for UC1-10

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. pay validation</td>
<td>Action noun DT</td>
<td>validate → AUT</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control otherwise constraint</td>
<td>denotes branching</td>
<td>SEL or guard (HM-2 or HM-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>denotes branching guard (HM-3) for validation status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“again” and “again”</td>
<td>implies repetition</td>
<td>LOOP adapter (HM-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data payment</td>
<td>Data amount paid</td>
<td>HM-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIN</td>
<td>Data PIN value</td>
<td>HM-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>Data constraint</td>
<td>HM-4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. The Cashier and the Customer try again and again.
2. Otherwise the Cashier requires the Customer to pay cash.

**Step-1 & Step-2** See Table C.9.

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.9.

**Step-4** This partial architecture has already been modelled in the existing system architecture. The only missing part is the constraint when the payment type is changed to cash payment. We assume that this part is handled by the AUT component, hence, is not modelled.

**Issue and discussion** In the current semantics of the X-MAN, breaking a loop is not defined. In order to ensure that the system terminates, the use of break and its consequences must be carefully considered.
APPENDIX C

Requirement-UC1-11

11. In step 6: Inventory not available.

1. The System caches each sale and writes them into the Inventory as soon as it is available again.

Table C.10: Summary of Steps 1 & 2 for UC1-11

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extraction elements</td>
<td>Extraction category</td>
<td>Map to Component-based Constructs</td>
</tr>
<tr>
<td>Comp. check available</td>
<td>Descriptive expression</td>
<td>isAvailable (\rightarrow) INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td>cache DT</td>
<td>cacheST (\rightarrow) ST</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>update DT</td>
<td>update (\rightarrow) INV</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control ‘and’ conjunction</td>
<td>PIPE (HM-2)</td>
<td>PIPE(isAvailable (\rightarrow) INV, \text{GUARD}(available=N) (\rightarrow) cacheST (\rightarrow) ST, \text{GUARD}(available=Y) (\rightarrow) update (\rightarrow) INV)</td>
<td></td>
</tr>
</tbody>
</table>

Data available Data boolean(Y/N) HM-4

Step-1 & Step-2 See Table C.10.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.10(a).

(a)UC1-R11

(b)Incremented architecture

Fig. C.10: UC1-R11

Step-4 The composition for UC1-R11 is added to the current system architecture as illustrated in Fig. C.10(b).
Issue and discussion Observe that the partial composition is composed in between the existing components as a result of refactoring task. Without refactoring, we cannot guarantee that the formerly modelled behaviours are preserved.

C.1.13 Use Case 2 – Manage Express Checkout

Requirement-UC2-1a

2. The considered Cash Desk is in normal mode and has just finished a sale which matches the condition of an express checkout sale. Now 50% of all sales during the last 60 minutes fulfil the condition for an express checkout.

(a) This Cash Desk, which has caused the achievement of the condition, is switched into express mode.

<table>
<thead>
<tr>
<th>Table C.11: Summary of Steps 1 &amp; 2 for UC2-1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extraction result</strong></td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Comp.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Control</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.11.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.11.

Step-4 As there is not enough information to compose the current architecture into the incremented architecture, the architecture is deferred.
**Issue and discussion** Each time a constraint data is identified, a corresponding implicit control to check the constraint must be allocated. This control can later be mapped into a guard adapter or a SEL whenever the condition is met.

**Requirement-UC2-1b**

2(b) Furthermore the corresponding Light Display (LD) is switched from black to green to indicate the Cash Desk’s express mode.

<table>
<thead>
<tr>
<th>Table C.12: Summary of Steps 1 &amp; 2 for UC2-1b</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extraction result</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Comp.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Control</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.12.

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.12.
C.1. THE TRADING SYSTEM (COCOME)

(a) UC2-R1b
(b) Incremented architecture

Step-4 We continue this increment from the previous requirement.

**Issue and discussion** Not applicable.

**Requirement-UC2-1c**

2(c) Paying by credit card is not possible any more.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. “paying card...”</td>
<td>“by” implies cash payment only</td>
<td>“setPaymentType” (PT)</td>
<td>HM-1B</td>
<td></td>
</tr>
<tr>
<td>Control constraint</td>
<td>check mode</td>
<td>Guard for checking mode</td>
<td>GUARD(mode=EM)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>paymentType</td>
<td>mode</td>
<td>cash</td>
<td>HM-4</td>
</tr>
<tr>
<td>Data</td>
<td>paymentType</td>
<td>data</td>
<td>EM only</td>
<td>HM-4</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.13. This statement contains constraints a single computation and data only.

**Step-3** The partial architecture for this requirement is demonstrated in Fig. C.13(a).

**Step-4** The composition for UC2-R1c is added to the current system architecture as illustrated in Fig. C.13(b).
**Issue and discussion** Based on the extracted information, the next step is to compose the architecture with the previous increments. This decision is based on the requirement that card payment is not available for express mode. Hence, only cash is allowed.

**Requirement-UC2-1d**

2(d) The maximum of items per sale is reduced to 8 and only paying by cash is allowed.

<table>
<thead>
<tr>
<th>Table C.14: Summary of Steps 1 &amp; 2 for UC2-1d</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extraction result</strong></td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Comp. “item per sale”</td>
</tr>
<tr>
<td>Control maximum of items</td>
</tr>
<tr>
<td>payment constraint</td>
</tr>
<tr>
<td>Data paymentType</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.14. The identify Item transaction has already been modelled in UC1. As we identify constraint data that are the payment...
type and the counter, they need to be checked by adapters (either or both, a guard or a loop). In this case, both the counter and the payment type can be used to set the number of repetitions and the constraint of a loop.

**Step-3** The partial architecture for this requirement is demonstrated in Fig. C.14(a).

![Diagram](Image)

(a)UC2-R1d  
(b)Part of Incremented Architecture

Fig. C.14: UC2-R1d

**Step-4** The composition for UC2-R1d is added to the current system architecture as illustrated in Fig. C.14(b).

**Issue and discussion** Another guard for checking the II process if the transaction is in express mode must also be added. This is important because when the transaction is in express mode, the item identifier process can be repeated not more than 8 times, whereas there is no constraint in normal mode.

When a composition is adapted, we assume the former part of the incremented architecture is de-composed and is being replaced by a new adapted composition. Then only the newly replaced composition will be composed to the former composition point.

**Requirement-UC2-2a**

2 The Cash Desk is in express mode and the Cashier decides to change back into normal mode.

(a) The Cashier presses the button Disable Express Mode.
Table C.15: Summary of Steps 1 & 2 for UC2-2a

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>“is in express mode”</td>
<td>DE</td>
<td>getMode → CD</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“change back to...”</td>
<td>DE</td>
<td>setNM → MODE</td>
<td>HM-1B</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>‘and’ implies ordering</td>
<td>PIPE(HM-2)</td>
<td>PIPE(getMode → CD, GUARD(mode=EM) → setNM → NM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>mode</td>
<td>Constraint data</td>
<td>NM</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

(a) UC2-R2a  (b) Part of Incremented Architecture

Fig. C.15: UC2-R2a

**Step-4** Since the CD component has already been modelled in the previous increment, the only composition that is required is the MODE component. Hence, the composition for UC2-R2a is added to the end of the relevant branches of the current system architecture as illustrated in Fig. C.15(b).

**Issue and discussion** Not applicable.

**Requirement-UC2-2b**

2(b) The colour of the Light Display is changed from green to black colour.

**Step-1 & Step-2** See Table C.16.
Table C.16: Summary of Steps 1 & 2 for UC2-2b

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. changed to DT</td>
<td></td>
<td></td>
<td>setBlk → LD</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control `from' implies ordering</td>
<td></td>
<td></td>
<td>PIPE</td>
<td>HM-2</td>
</tr>
<tr>
<td>Data color Store data black, green</td>
<td></td>
<td></td>
<td>black, green</td>
<td>HM-4</td>
</tr>
</tbody>
</table>

**Step-3** As only a single computation is identified, no partial architecture is produced. However, for illustration purpose, see Fig. C.16(a).

![Diagram](image)

(a)UC2-R2b (b)Part of Incremented Architecture for UC2-R2b

**Fig. C.16: UC2-R2b**

**Step-4** The composition for UC2-R2b is added to the current system architecture as illustrated in Fig. C.16(b).

**Issue and discussion** To ensure that the assigned component is added to the correct composition point, a guard is also required to check that the mode is normal.

**Requirement-UC2-2c**

2(c) Cash and also card payment is allowed and the Customer is allowed to buy as many goods as he likes.

**Step-1 & Step-2** See Table C.17.

**Step-3** The partial architecture for this requirement is demonstrated in Fig. C.17(a).
Table C.17: Summary of Steps 1 & 2 for UC2-2c

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. cash payment</td>
<td>Action noun payCash</td>
<td>CP HM-1B</td>
<td>payCash → CP HM-1B</td>
<td></td>
</tr>
<tr>
<td>card payment</td>
<td>Action noun payCard</td>
<td>CDP HM-1B</td>
<td>payCard → CDP HM-1B</td>
<td></td>
</tr>
<tr>
<td>Control ‘and’ loop</td>
<td>does not imply ordering, but a selection instead</td>
<td>SEL or Guard LOOP HM-2, HM-3</td>
<td>LOOP HM-3</td>
<td></td>
</tr>
<tr>
<td>Data goods</td>
<td>counter noOfItem</td>
<td>complete implicit constraint to end the scanning process</td>
<td>Y/N HM-4</td>
<td></td>
</tr>
</tbody>
</table>

(a)UC2-R2c: (c)Part of Incremented Architecture  
Fig. C.17: UC2-R2c

**Step-4** The composition for UC2-R2c has already been modelled in the current system architecture as illustrated in Fig. C.17(b). The loop has been modelled for identifying items in another branch.

**Issue and discussion** The conjunction ‘and’ here does not mean ordering, it implies that both the payment types are available for customers. Hence,
we map that to a SEL instead. In Fig. C.17(b), the SEL is renamed into paymentType.

C.1.14 Use Case 3 – Order Products

Requirement-UC3-1

1. A list with all products and a list with products running out of stock are shown.

<table>
<thead>
<tr>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. “list with all products”</td>
<td>Descriptive expression</td>
<td>viewStock → INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td>“list .out of stock”</td>
<td>Descriptive expression</td>
<td>outOfStock → INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td>implicit authentication</td>
<td>DT</td>
<td>aut → AUT</td>
<td>only the Store Mgr can access this function.</td>
</tr>
<tr>
<td>Control constraint</td>
<td>aut status</td>
<td>guard</td>
<td>HM-3</td>
</tr>
<tr>
<td>‘and’</td>
<td>denotes both PIPE (HM-2)</td>
<td>PIPE (aut → AUT, GUARD(Y) → INV)</td>
<td></td>
</tr>
<tr>
<td>constraint to switch mode</td>
<td>Guard(HM-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data status</td>
<td>Constraint data</td>
<td>Y/N (valid/invalid)</td>
<td>HM-4</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.18. How do we extract authorisation computation for the store manager? Although there is no explicit computation to be extracted for such an authorisation, clearly there is a need for that. This additional information is included in the introduction part of the use case. Hence, an authentication component to authenticate the Store Manager is added.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.18.

Step-4 At this point, there is not enough information to compose the current architecture into the incremented architecture. Hence, the composition to the incremented architecture is deferred.

Issue and discussion Not applicable.
Requirement-UC3-2

2. The Store Manager chooses the product items to order and enters the corresponding amount.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. order</td>
<td>DT</td>
<td>viewStock \rightarrow INV</td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data amount</td>
<td>Data</td>
<td>order amount</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.19. Note that the ‘and’ conjunction here is not identified as a control. Here, it can be handled as internal execution of the order computation. Once the order is invoked, it will prompt for the corresponding input.

**Step-3** No partial architecture for this requirement as only a single computation being extracted.

**Step-4** We continue this increment from the previous requirement. The composition for the assigned component which is extracted from UC3-R2 is added to the current system architecture as illustrated in Fig. C.19.
Issue and discussion Only view stock is exposed and out of stock is considered as an internal computation. The incremented architecture has already been refactored. Otherwise we need two guards checking the same constraint that is the authentication status for INV and ODR branches. In order to represent the correct behaviour, we bring forward the guard so that the checking is done once, instead of twice.

Requirement-UC3-3

3. The Store Manager presses the button Order at the Store Client’s GUI.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction results</th>
<th>Mapping results</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. order</td>
<td>user interaction</td>
<td>order → INV</td>
<td>HM-1B</td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.20.

Step-3 No partial architecture for this requirement as only a single computation is extracted.

Step-4 We continue this increment from the previous requirement.

Issue and discussion Not applicable.
Table C.21: Summary of Steps 1 & 2 for UC3-4

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>place order</td>
<td>DT</td>
<td>order → ODR</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>generate</td>
<td>DT</td>
<td>generateOrderId</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shown</td>
<td>DT</td>
<td>display → ODR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>‘and’ conjunction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>supplier, order, orderId</td>
<td>Store values</td>
<td>Data</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

**Requirement-UC3-4**

4. The appropriate suppliers are chosen and orders for each supplier are placed. An order identifier is generated for each order and is shown to the Store Manager.

**Step-1** See Table C.21. We assume when placing an order, the supplier will also be decided.

**Step-2** As all the extracted computations are assigned to the same component, there is no need of any composition connectors. Indirectly, we assume that all these computations are internally dealt with by the ODR component.

**Step-3** No partial architecture for this requirement as only a single component is extracted.

![Fig. C.21: Part of the incremented architecture for UC3-R4](image-url)
C.1. THE TRADING SYSTEM (COCOME)

Step-4 The only exposed computation is the order computation. During selection of ODR component, we have to ensure the generate order id and display computations are also provided.

In order to compose the current partial architecture with the incremented system architecture, the only possible way is using a top-level SEL. See Fig. C.21.

Issue and discussion Not applicable.

C.1.15 Use Case 4 – Receive Ordered Products

Requirement-UC4-1

1. Ordered products arrive at the stock attached with an order identifier which has been assigned during the ordering process.

Step-1 This is not a functional requirement. Hence, all the following steps for this increment are skipped.

Step-2 Not applicable.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC4-2

2. The Stock Manager checks the delivery for completeness and correctness.

Step-1 This is manually done by the Stock Manager and is not a functional requirement. Hence, all the following steps for this increment are skipped.

Step-2 Not applicable.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.
Table C.22: Summary of Steps 1 & 2 for UC4-3

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. roll</td>
<td>user interaction</td>
<td>implicit DT</td>
<td>receive → ODR</td>
<td>HM-1B</td>
</tr>
<tr>
<td>Stock Manager</td>
<td></td>
<td></td>
<td>aut → AUT</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>implicit pass the aut status</td>
<td>PIPE HM-2</td>
<td>PIPE (aut → AUT, GUARD (stkMgr) → roll → ODR)</td>
<td></td>
</tr>
<tr>
<td>constraint check authentication status</td>
<td>Guard (HM-3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data status</td>
<td>Constraint data</td>
<td>Y/N (valid/invalid)</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

Requirement-UC4-3

3. In the case of correctness, the Stock Manager enters the order identifier and presses the button Roll in received order.

Step-1 & Step-2 See Table C.22. Although there is no explicit computation to be extracted for such an authorisation, clearly there is a need for that. This additional information is included in the introduction part of the use case. Hence, an authentication component to authenticate the Stock Manager is added.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.22.

![Fig. C.22: UC4-R3](image)

Step-4 At this point, there is no composition point to be used. Hence, the composition to the incremented architecture is deferred.

Issue and discussion Not applicable.
C.1. THE TRADING SYSTEM (COCOME)

Requirement-UC4-4

4. The Trading System updates the Inventory.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. up date</td>
<td>DT</td>
<td>update (\rightarrow) INV</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>updatedOrderData</td>
<td>received order</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.23.

Step-3 As only a single computation is extracted, there is no partial architecture for this requirement.

Fig. C.23: UC4-R4

(a)UC4-R4  (b)Part of the incremented architecture

Step-4 The composition for UC4-R4 is added to the current system architecture using an existing SEL as depicted in Fig. C.23(b). This is considered as the current incremented system architecture produced so far.

Issue and discussion The partial architecture has already refactored the same guards and bring upfront the guards so that the derived architecture will reduce the number of guards.
Requirement-UC4-5

5. In step 2: Delivery not complete or not correct. The products are sent back to the supplier and the Stock Manager has to wait until a correct and complete delivery has arrived. This action is not recognised by the System.

Step-1 This is not a functional requirement. Hence, all the following steps for this increment are skipped.

Step-2 Not applicable.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

C.1.16 Use Case 5 – Show Stock Reports

Requirement-UC5-1

1. The Store Manager enters the store identifier and presses the button Create Report.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comp.</td>
<td>create</td>
<td>create -&gt;RG</td>
</tr>
<tr>
<td></td>
<td>“Store Manager enters..”</td>
<td>aut -&gt;AUT</td>
</tr>
<tr>
<td></td>
<td>DE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DE</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>‘and’</td>
<td>PIPE HM-2</td>
</tr>
<tr>
<td></td>
<td>conjunction, implies ordering</td>
<td>PIPE(authenticate -&gt;AUT, GUARD(valid) -&gt;create -&gt;RG)</td>
</tr>
<tr>
<td></td>
<td>aut status</td>
<td>Guard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HM-3</td>
</tr>
<tr>
<td>Data</td>
<td>username, pwd</td>
<td>Constraint data authentication purpose</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step-1 & Step-2 See Table C.24. Only a single component identified explicitly from the requirement. However, we assume the authentication process must be performed prior to the report creation. The authentication process is implied from the COCOME’s introduction.

Step-3 Fig. C.24 depicts the partial architecture for this requirement.

![Partial Architecture](image.png)

**Fig. C.24: UC5-R1**

Step-4 At this point, no composition point in the system architecture can be used. Hence, the composition to the incremented architecture is deferred.

**Issue and discussion** Not applicable.

**Requirement-UC5-2**

2. A report including all available stock items in the store is displayed.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. display</td>
<td>DT</td>
<td>display →RG</td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>internal invocation</td>
</tr>
<tr>
<td>Data</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.25. Only a single computation is extracted. This computation is treated as an internal computation, which can be invoked by the exposed interface.

Step-3 There is no partial architecture for this requirement.

Step-4 The component is added to the incremented system architecture (see Fig. C.25).

**Issue and discussion** Not applicable.
C.1.17 Use Case 6 – Show Delivery Reports

Requirement-UC6-1

1. The Enterprise Manager enters the enterprise identifier and presses the button Create Report.

Table C.26: Summary of Steps 1 & 2 for UC6-1

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extracted elements</td>
<td>Extraction category</td>
</tr>
<tr>
<td>Comp.</td>
<td>create DT</td>
<td>DE</td>
</tr>
<tr>
<td></td>
<td>“Enterprise Manager enters...”</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>‘and’ ordering</td>
<td>PIPE</td>
</tr>
<tr>
<td></td>
<td>constraint aut status</td>
<td>Guard</td>
</tr>
<tr>
<td>Data</td>
<td>username, pwd</td>
<td>Constraint data authentication purpose</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.26.

Step-3 The partial architecture for this requirement is shown in Fig. C.26.
Step-4 As there is not enough information to compose the current architecture into the incremented architecture, the architecture is deferred until a valid composition point is available.

**Issue and discussion** Not applicable.

**Requirement-UC6-2**

2. A report which informs about the mean times is generated.

<table>
<thead>
<tr>
<th>Table C.27: Summary of Steps 1 &amp; 2 for UC6-2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extraction result</strong></td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Comp. calculate DT</td>
</tr>
<tr>
<td>Control</td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.27. Only a single computation is extracted. This computation is treated as an internal computation, which can be invoked by the exposed interface.

**Step-3** There is no partial architecture for this requirement.

**Step-4** The assigned component is composed to the incremented system architecture. This design is considered as the current system diagram. See Fig. C.27.

**Issue and discussion** Not applicable.
C.1.18 Use Case 7 – Change Price

Requirement-UC7-1

1. The System presents an overview of all available products in the store.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. presents</td>
<td>DT</td>
<td>view → INV</td>
</tr>
<tr>
<td>authenticate</td>
<td>DE (intro)</td>
<td>aut → AUT</td>
</tr>
<tr>
<td>Control constraint</td>
<td>aut status</td>
<td>Guard</td>
</tr>
<tr>
<td>implicit ordering</td>
<td>PIPE</td>
<td>PIPE(authenticate \rightarrow AUT, \text{GUARD}(\text{valid}) \rightarrow \text{view} \rightarrow \text{INV})</td>
</tr>
<tr>
<td>Data</td>
<td>username, pwd</td>
<td>authentication</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.28. An authentication computation by the Store Manager, is added.

**Step-3** The partial architecture for this requirement is shown in Fig. C.28.

**Step-4** See Fig. C.28.

**Issue and discussion** Not applicable.
Requirement-UC7-2

2. The Store Manager selects a product item and changes its sales price.

Table C.29: Summary of Steps 1 & 2 for UC7-2

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. select</td>
<td>user interaction</td>
<td>viewItem \rightarrow INV</td>
<td>HM-1B</td>
</tr>
<tr>
<td>Comp. search</td>
<td>implicit DT</td>
<td>search \rightarrow INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data</td>
<td>item, price</td>
<td>store data</td>
<td>Data</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.29.

**Step-3** As the extracted computations belong to the same component, no partial architecture for this requirement is derived.

**Step-4** Not applicable.

**Issue and discussion** Not applicable.

Requirement-UC7-3

3. The Store Manager commits the change by pressing ENTER.

**Step-1 & Step-2** See Table C.30.

**Step-3** As only a single component is identified, in effect, no partial architecture for this requirement is derived.

**Step-4** The incremented architecture for UC7-3 is shown in Fig. C.29.
Table C.30: Summary of Steps 1 & 2 for UC7-3

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. change</td>
<td>user interaction</td>
<td>update → INV</td>
<td>HM-1B</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data item, price store data Data HM-4

Fig. C.29: Part of the incremented architecture for UC7-R3

**Issue and discussion** In this case, only view computation is exposed once invoked, the user will be allowed to do the changes and the changes will be updated to the system. All these computations are handled by INV component.

**C.1.19 Use Case 8 – Product Exchange (on low stock) Among Stores**

**Requirement-UC8-1**

1. A certain product of the Store runs out.

**Step-1 & Step-2** See Table C.31.

**Step-3** There is no partial architecture for this requirement.

**Step-4** Not applicable.

**Issue and discussion** This checking stock computation is invoked automatically. Once a store client is running, the Store Server checks the stock.
Table C.31: Summary of Steps 1 & 2 for UC8-1

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. runs</td>
<td>DT</td>
<td>checkStock → STK</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Data product</td>
<td>Data</td>
<td>productId</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

**Requirement-UC8-2**

2. The Store Server recognizes low stock of the product.

Table C.32: Summary of Steps 1 & 2 for UC8-2

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. recognizes</td>
<td>DT</td>
<td>checkLowStock → STK</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Data product</td>
<td>Data</td>
<td>productId</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.32.

**Step-3** There is no partial architecture for this requirement.

**Step-4** Not applicable.

**Issue and discussion** The low stock of each product is formerly set in the database and the checking for low stock will be based on the setting.

**Requirement-UC8-3**

3. The Store Server sends a request to the Enterprise Server (including an identification of the shortage products, and a Store id).
Table C.33: Summary of Steps 1 & 2 for UC8-3

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. identification Action noun</td>
<td>checkLowStock → STK</td>
<td>HM-1A</td>
<td>PIPE</td>
<td>PIPE(checkLowStock → STK, sendRequest → SS)</td>
</tr>
<tr>
<td>send request DT</td>
<td>sendRequest → SS</td>
<td>HM-1A</td>
<td>PIPE</td>
<td>PIPE(checkLowStock → STK, sendRequest → SS)</td>
</tr>
<tr>
<td>Control implicit ordering</td>
<td>PIPE</td>
<td>HM-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data product store ID</td>
<td>Data productId, storeId</td>
<td>HM-4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.33.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.30.

![Fig. C.30: UC8-R3](image)

Step-4 Not applicable.

Issue and discussion Note that the piping execution here is to represent communication between servers.

Requirement-UC8-4

4. The Enterprise Server triggers all Stores that are “nearby” (e.g. 300 km) the requiring store, to flush their local write caches. So the Enterprise Server database gets updated by the Store Server.

Step-1 & Step-2 See Table C.34.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.31.

Step-4 Not applicable.

Issue and discussion Not applicable.
### Table C.34: Summary of Steps 1 & 2 for UC8-4

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. trigger</td>
<td>DT</td>
<td>trigger → ES</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>DT</td>
<td>update → SS</td>
<td>HM-1A</td>
<td></td>
</tr>
<tr>
<td>Control Implicit ordering</td>
<td>PIPE</td>
<td>PIPE(trigger → ES, update → SS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data product</td>
<td>Data</td>
<td>productId</td>
<td>HM-4</td>
<td></td>
</tr>
<tr>
<td>storeId</td>
<td>Data</td>
<td>StoreId</td>
<td>HM-4</td>
<td></td>
</tr>
<tr>
<td>reqStoreId</td>
<td>Data</td>
<td>ReqStoreId</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

(a)UC8-R4  (b)Incremented architecture

Fig. C.31: UC8-R4

### Requirement-UC8-5

5. The Enterprise Server does a database look-up for the required products to get a list of products (including amounts) that are available at providing Stores.

#### Step-1 & Step-2
See Table C.35.

#### Step-3
There is no partial architecture for this requirement.

Fig. C.32: Incremented architecture for UC8-R5
Table C.35: Summary of Steps 1 & 2 for UC8-5

<table>
<thead>
<tr>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Extracted elements</td>
</tr>
<tr>
<td>Comp. look-up</td>
<td>DT</td>
</tr>
<tr>
<td>Control implicit ordering</td>
<td>PIPE</td>
</tr>
<tr>
<td>Data product store amount</td>
<td>Data Data Data</td>
</tr>
</tbody>
</table>

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC8-6

6. The Enterprise Server applies the “optimisation criterion” (specified above) to decide as to whether it is meaningful to transport the shortage product from one store to another (heuristics might be applied to minimize the total costs of transportation). This results in a list of products (including amounts) per providing store that have to be delivered to the requiring Store.

Table C.36: Summary of Steps 1 & 2 for UC8-6

<table>
<thead>
<tr>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Extracted elements</td>
</tr>
<tr>
<td>Comp. optimisation</td>
<td>Action noun</td>
</tr>
<tr>
<td>Control implicit ordering</td>
<td>PIPE</td>
</tr>
<tr>
<td>Data product store product information</td>
<td>HM-4</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.36. We assume that this is an internal invocation performed by the ES.
C.1. THE TRADING SYSTEM (COCOME)

Step-3 There is no partial architecture for this requirement.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC8-7

7. The Store Server, initially sending the recognition of the shortage product, is provided with the decision of the Enterprise Server.

   (a) The required product is marked as incoming.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>optimise DT</td>
<td>sendRecognition(storeId, HM-1A status) →ES</td>
</tr>
<tr>
<td></td>
<td>marked DT</td>
<td>mark(status) →SS</td>
</tr>
<tr>
<td>Control</td>
<td>implicit invocation</td>
<td>PIPE(sendRecognition →ES, mark →SS)</td>
</tr>
<tr>
<td>Data</td>
<td>incoming</td>
<td>Data incoming(status)</td>
</tr>
</tbody>
</table>

Table C.37: Summary of Steps 1 & 2 for UC8-7

Step-1 & Step-2 See Table C.37.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.33.

Fig. C.33: Incremented architecture for UC8-R7

Step-4 Not applicable.

Issue and discussion Not applicable.
Requirement-UC8-8

8. The Store Server of a nearby Store is provided with information that it has to deliver the product.

(a) The required product is marked as unavailable in the Store.

Table C.38: Summary of Steps 1 & 2 for UC8-8

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>delivery</td>
<td>DT</td>
<td>notifyDelivery(storeId, prodId, amount) (\rightarrow) ES</td>
<td>HM-1A</td>
</tr>
<tr>
<td></td>
<td>marked</td>
<td>DT</td>
<td>mark(status) (\rightarrow) SS</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>implicit invocation</td>
<td>ordering</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data</td>
<td>unavailable</td>
<td>Data</td>
<td>availability status</td>
<td>-</td>
</tr>
</tbody>
</table>

**Step-1 & Step-2** See Table C.38.

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.34(a).

![Fig. C.34: Incremented architecture for UC8-R8](image)

**Step-4** The current system architecture is illustrated in Fig. C.34(b).

**Issue and discussion** Not applicable.

Requirement-UC8-E1

E1. The Enterprise Server is not available: The request is queued until the Enterprise Server is available and then is sent again.

**Step-1 & Step-2** See Table C.39.
### Table C.39: Summary of Steps 1 & 2 for UC8-E1

<table>
<thead>
<tr>
<th>Description</th>
<th>Extracted elements</th>
<th>Extraction category</th>
<th>Map to Component-based Constructs</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. “is available”</td>
<td>DE</td>
<td>Extracted elements</td>
<td>Extracted elements</td>
<td>Extraction category</td>
</tr>
<tr>
<td>queue</td>
<td>DT</td>
<td>queue</td>
<td>ES HM-1A</td>
<td>queue → SS</td>
</tr>
<tr>
<td>send</td>
<td>DT</td>
<td>send</td>
<td>SS HM-1A</td>
<td>sendReq → SS</td>
</tr>
<tr>
<td>Control</td>
<td>‘and’ ordering</td>
<td>PIPE</td>
<td>PIPE(chkAvailable → ES, queue → SS)</td>
<td>Guard(HM-3)</td>
</tr>
<tr>
<td>constraint</td>
<td>availability status</td>
<td>available(Y/N)</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image)

**Fig. C.35: Incremented architecture for UC8-E1**

**Step-3** The partial architecture for this requirement is illustrated in Fig. C.35(a).

The current system architecture is illustrated in Fig. C.35(b).

**Issue and discussion** The X-MAN component model accommodates for single-threaded execution; a loop to queue for waiting for the ES to be available cannot be explicitly modelled. In this case, we assume the SS has an implicit server computation to perform such a task.

### C.1.20 Extension on UC8 - Remove Incoming Status

**Requirement-UC8-Ext1**

1. The products arrive at the stock of the Store.

**Step-1** This is not a functional requirement. Hence, all the following steps for this increment are skipped.

**Step-2** Not applicable.
Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC8-Ext2

2. For all arriving products the Stock Manager counts the incoming amount.

Step-1 This is not a functional requirement. Hence, all the following steps for this increment are skipped.

Step-2 Not applicable.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC8-Ext3

3. For every arriving product the Stock Manager enters the identifier and its amount into the Store Client.

Table C.40: Summary of Steps 1 & 2 for UC8-Ext3

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Map to Component-based Constructs</th>
<th>Mapping result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>enter DT</td>
<td>enterItemId →INV</td>
<td>HM-1B</td>
</tr>
<tr>
<td>Store Manager</td>
<td>implicit</td>
<td>aut →AUT</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control</td>
<td>implicit ordering</td>
<td>PIPE</td>
<td>PIPE(aut →AUT, enterItemId →INV)</td>
</tr>
<tr>
<td>constraint</td>
<td>authentication status</td>
<td>Guard(HM-3)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>status Constraint data</td>
<td>aut status(Y/N)</td>
<td>HM-4</td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.40.

Step-3 The partial architecture for this requirement is illustrated in Fig. C.36(a).
Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-UC8-Ext4

4. The system updates the Inventory.

<table>
<thead>
<tr>
<th>Description</th>
<th>Extraction result</th>
<th>Mapping result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp. update</td>
<td>DT</td>
<td>update (\rightarrow) INV</td>
<td>HM-1A</td>
</tr>
<tr>
<td>Control implicit</td>
<td>ordering</td>
<td>previous increment</td>
<td></td>
</tr>
<tr>
<td>constraint valid status</td>
<td>Guard(HM-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data product data(implicit)</td>
<td>product to be updated</td>
<td>HM-4</td>
<td></td>
</tr>
</tbody>
</table>

Step-1 & Step-2 See Table C.41.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion We assume the update is internally invoked by the exposed interface’s computation.
Fig. C.37: COCOME Architecture Before Refactoring

Fig. C.38: Architecture Refactoring
C.1.21 Architecture Refactoring of the COCOME

After all the incremental steps, the following step is the refactoring step. The main purpose of refactoring is to simplify the structure. The refactoring tasks have been discussed in Chapter 7. To illustrate the refactorings of the COCOME example, the discussion will be provided according to the label AR-1 to AR-6 in Fig. C.38.

C.1.22 AR-1

From label AR-1, both the same guards that are checking the same constraint i.e. normal or express modes, are combined. In effect, the guards are brought forward so that they can check the constraints once before the execution of setMode and change the light display. See Fig. C.39(a). However, we can still see that the guards that are mutually exclusive, can also be refactored. Hence, Fig. C.39(b) depicts the following refactoring task to replace them with a SEL. As a result, the structure becomes more transparent and understandable.

![Fig. C.39: AR-1](image)

C.1.23 AR-2

In this case, both the branches can be combined using a loop adapter. See Fig. C.40(a). The loop can be designed in such a way that all the constraints are encapsulated in it. This loop checks the looping constraint i.e. complete, and also the modes i.e. express or normal.

C.1.24 AR-3

Fig. C.41(a) demonstrates the original structure before refactoring. For this case, we simply combine both of the guards because they are checking the same constraint. To ensure the same behaviour is modelled, one guard (instead of two) is brought upfront. Hence, the checking will only be performed once. See Fig. C.41(b).
C.1.25 AR-4

In Fig. C.42(a), we can combine the same guards i.e. checking the item’s availability. As a result of this, we need to bring the combine guard upfront so that the checking is done once, instead of twice (see Fig. C.42(b)).

C.1.26 AR-5

Fig. C.43(a) demonstrates the original structure before refactoring. Here, the guards are mutually exclusive. Thus, we map them into a SEL. See Fig. C.43(b).
Accordingly, in Fig.C.43(b) the SEL will be connected to the existing PIPE in the upper hierarchy level.

![Diagram](image_url)

(a) Before (b) After refactoring

Fig. C.43: AR-5

C.1.27 AR-6

Fig. C.44(a) demonstrates the original structure before refactoring. In this figure, all the guards that are checking the authentication status of the managers can be combined. We mapped them to a SEL. The selection criterion is based on the valid status and the transaction type. See Fig.C.44(b). The advantage of restructuring is to reduce complexity. In such an attempt, in this particular case, we have shown that the unnecessary guards and composition connectors can be reduced. As a result, the structure becomes more readable.

![Diagram](image_url)

(a) Before (b) After Refactoring

Fig. C.44: AR-6

The complete architecture after refactoring is depicted in Fig. C.45. Observe that this refactored architecture has a simplified and clearer structure compared to the original architecture (see Fig. C.37).

The final step during finalisation is to fix all the open composition points. Now, all the open composition connectors are changed to solid interface, which denotes no composition points are available for further composition (see Fig. C.46).
Fig. C.45: COCOME Architecture After Refactoring

Fig. C.46: Finalised Architecture
C.1.28 Summary

In this case study, the all the defined steps in the approach to constructing component-based systems have been demonstrated, applied and presented. The primary concern is to identify elements of component-based systems as defined in the X-MAN component model, namely computations, control and data. Following this premise, a complete COCOME example has been demonstrated as to apply all the heuristics and design decisions defined in this thesis. It has also been elucidated that the identified computations, control and data can be used to guide us in constructing the complete component-based system. Although this approach is basically heuristic, and requires human guidance and decision making, we believe this is possible because the underlying component model provides a way to realise such an approach.
C.2 The Video Store System (VSS)

C.2.1 Introduction

This Video Store example is taken from [LSB98]. To simplify the documentation, each step will be labelled as Step-1 to Step-5 accordingly. The documentation of the execution of the VSS case study using the incremental approach will be grounded on the following steps:

Step-1 Extracting elements of component-based systems from requirements;

Step-2 Mapping of the extracted elements to X-MAN elements;

Step-3 Creating a partial architecture;

Step-4 Composing the partial architecture with the existing system architecture;

Step-5 Finalising the system architecture.

Note that Step-5 is only applicable after the final requirement has been executed. An item for any issue and discussion is also added.

C.2.2 Incremental Execution of the VSS

Requirement-1

1. In the initial state of the video system the main menu is displayed. From the main menu the clerk can choose one of the following options:

   1. Rent a tape;
   2. Return tapes;
   3. Insert new customer;
   4. Insert new video;
   5. Change customer data;
   6. Change video data;
   7. Delete video;
   8. Delete customer.
Step-1 & Step 2 Identify computations i.e. rent tape, return tape, insert new customer, insert new video, change customer data, change video data, delete video, delete customer and exit. For each computation, we decide whether we want to group them into meaningful components or set individual computations as independent components. For control, we identify a selection of the above computations based on the control term option. Hence, we map this execution to a selector here.

Step-3 The partial architecture for R1 is as shown in Fig. C.47.

Fig. C.47: Result of mapping for R1

Step-4 Since this is the first partial architecture, this is considered as the initial system diagram.

Issue and discussion Not applicable.

Requirement-2

2. The system keeps a video inventory record for each tape given attributes and the current status of it.

Step-1 & Step 2 Video Inventory Record (VIR) is a database element. This is also a conceptual component because there is no computation that we can extract yet at this point.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-3

3. The system keeps a rental transaction record for each customer giving out information and currently rented tapes for each customer.
Step-1 & Step 2 Rental Transaction Record (RTR) is a database element. This can be considered as a conceptual component because there is no computation that we can extract with regard to RTR at this point.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-4

R4. The customer has his ABC card with him. The account number of a customer is read with the bar code reader from the clerk to retrieve rental transaction record.

Step-1 & Step 2 Identify computations i.e. read card, search RTR and display. Use a pipe to pass the account number read from the card reader in order to retrieve the rental transaction record by the RTR component.

Step-3 The partial architecture for R4 is as shown in Fig. C.48(a).

![Partial Architecture for R4](image)

(a) R4 (b) Incremented architecture

Fig. C.48: R4

Step-4 When we decompose a component, we can either decompose only or decompose and extend the component by maintaining the previous identified computations. In this case, we decompose and extend the RT component by adding CR and RTR components and maintaining the rent tape computation. The result of incremental composition can be depicted in Fig. C.48(b).

Issue and discussion Not applicable.

Requirement-5

Description The customer doesn’t have his ABC card with him. The account number of a customer is entered with the keyboard from the clerk to retrieve the rental transaction record.

Input The account number is entered with the keyboard.
Processing Searching rental transaction record.

Output Display rental transaction record.

Step-1 & Step 2 Identify computations i.e. search RTR and display. However, these two computations have already been identified previously. We can put a constraint on the card reader component so that only customers with cards can be allowed to invoke this computation. Otherwise, the control flow will be passed to the RTR component.

Step-3 As both the extracted computations are assigned to be dealt with by the same component, there is no partial architecture for R5.

Step-4 The display computation is encapsulated in component RTR. The result of the incremental composition is depicted in Fig. C.49.

Issue and discussion The invocation to the RTR is done from the exposed interface of the component. In this case, via the search interface.

Requirement-6

Description Bar code id for each tape to be rented are entered.

Input Bar code id for each tape are entered with the bar code reader.

Processing Retrieving video inventory record about the tape.

Output Display video name and rental price.

Step-1 & Step 2 Identify computations i.e. read bar code ID, retrieve, display. We allocate read bar code ID computation into the Bar Code Reader component, and the rest into the Video Information Record (VIR) component. We need a pipe connector in order to pass the bar code data into the VIR component.

Step-3 The partial architecture for R6 is shown in Fig. C.50(a).

Step-4 Result of the incremental composition can be depicted in Fig. C.50(b).

Issue and discussion Not applicable.
Requirement-7

Description The maximal number of tapes that can be rented in one transaction is 20.

Input Bar code ids of tape is entered with the bar code reader.

Processing If 21 or more tapes are taken, rental is rejected.

Output Error message is displayed.

Step-1 & Step 2 The description contains only constraints which can be checked in the corresponding computation. Hence, the only computation that we identify here is to display the error message. However, this computation can be allocated in the same RTR component.

Step-3 Not applicable.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-8

Description When all tapes are entered the system computes the total.

Input Enter key is pressed on keyboard after the last tape was entered.

Processing Computation of the total due. The total is the sum of the past due fees, other fees and current video rental fees.

Output Total due.

Step-1 & Step 2 The description contains a computation to compute the total rental fee. This fee is calculated when all tapes have been entered. We extract compute and display change as DTs.
C.2. THE VIDEO STORE SYSTEM (VSS)

441

(a) R8

(b) Part of the Incremented Architecture for R8

Fig. C.51: R8

Step-3 The partial architecture can be modelled as in Fig. C.51(a).

Step-4 Result of the incremental composition can be depicted in Fig. C.51(b).

Issue and discussion Not applicable.

Requirement-9

Description The clerk collects the money from the customer and enters the amount into the system.

Input Amount of money that is given to clerk entered via keyboard.

Processing Compute change.

Output Display amount of change.

Step-1 & Step 2 The description contains a computation to compute the change and to display the result. In the same component, hence no flow is required.

Step-3 No partial architecture.

Step-4 The incremental system architecture can be designed as in Fig. C.52.

Issue and discussion Not applicable.

Requirement-10

Description When the clerk presses an order-complete option key (defined by the system) this rental is complete and the video inventory file is updated.

Input Clerk presses the order-complete option key.

Processing Update the video inventory file. Close rental transaction.
Output Video inventory file is updated. Rental transaction file is updated.

Step-1 & Step 2 The description contains computations to update the VIR once the rental process is completed. We need a guard to check if the rental process is complete. We assume when the clerk presses the order-complete key then the VIR will be updated.

Step-3 There is no partial architecture for this increment because only a single component and a guard are identified.

Step-4 The incremental system architecture can be designed as in Fig. C.53.

Issue and discussion Not applicable.

Requirement-11

Description After the rental is closed, the transaction is stored and printed.

Input Close the current rental.
**Processing** Store rental and print form that the customer has to sign. Return to initial state. Forms will be kept on file in the store for one month after the tapes are returned.

**Output** Printed form. Initial menu is displayed.

**Step-1 & Step 2** The description contains additional rental computations i.e. save, print, close. We add these computations to the RT component. Although we can extract the preposition ‘after’, but since these computations are provided by the same component, no control is therefore required.

**Step-3** These are internal computations provided by the RT component. Hence, there is no partial architecture for this increment.

![Diagram](image.png)

Fig. C.54: Part of the Incremented Architecture for R11

**Step-4** In this case, we model the RT component again as to capture the behaviour of the requirement. The incremental system architecture can be designed as in Fig. C.54.

**Issue and discussion** The invocations of these computations are managed by the RT component. From the context of the design, only the exposed interface is visible. Note that the RT components are labelled twice. This reflects the same instance being used in the architecture design. In order to maintain the tree structure, the model allows duplication of components to refer to the same component instance.

**Requirement-12**

**Description** Requirements for returning a tape. The video bar code is entered into the system.

**Input** Bar code of the video.

**Processing** Retrieve rental transaction record.
Output Display rental transaction record.

Step-1 & Step 2 The description contains computations i.e. read bar code, search RTR and display. Clearly, we can reuse the same BCR and RTR components. We retrieve the same BCR and RTR components. For simplicity purposes, we hide the other irrelevant components.

Step-3 See Fig. C.55(a) for the partial architecture.

Step-4 The incremental system architecture can be designed as in Fig. C.55(b).

Requirement-13

Description When the rental transaction record is retrieved the record of the video is marked with the date of return.

Input Bar code of the video rental.

Processing Rental transaction and video inventory record are retrieved. Mark date of return in the record.

Output Updated video inventory and rental transaction record.

Step-1 & Step 2 This statement describes the return tape computation. For this, we can refer to the same computation which we have identified previously in R1. The additional behaviour for marking the video with the date of return can optionally be added as a new computation or to document the additional information to its specification. In this case, we retrieve the same RTN component.

Step-3 See Fig. C.56(a) for the partial architecture.

Step-4 The incremental system architecture can be designed as in Fig. C.56(b).

Issue and discussion Not applicable.
C.2. THE VIDEO STORE SYSTEM (VSS)

Requirement-14

Description If past-due amounts are owed, they can be paid at this time; or the clerk can select the “order-complete” key which updates the rental with the return date and calculates past-due fees.

Input Payment or calculation of past-due fees.

Processing Updating rental transaction record. Go to initial state.

Output Updated rental transaction file.

Step-1 & Step 2 In this statement, we can directly identify update (must update RTR as well) and calculate fee computations. Both of these computations can be grouped into a single component i.e. the OVD component. Since the overdue process is not an independent process and it is significantly relevant to the return tape process, for the modelling purpose, we add the RTN component in the design. Based on the descriptive expression, we need a constraint to check whether the past-due amounts are owed or not. This can be realised by adding a guard that checks the constraint. In order to pass data between RTN and OVD components, we add a PIPE connector to compose them.

Step-3 The partial architecture for R14 can be modelled as in Fig. C.57(a).

Step-4 The incremental system architecture can be designed as in Fig. C.57(b).

Issue and discussion Not applicable.

Requirement-15

Description A new customer wants to rent tapes. The clerk enters all the necessary information, prints the bar code for the ABC card and glues it on a blank ABC card. Then this card is given to the customer.

Input Clerk enters the following information: Name, address and credit card information of the customer.
Processing Create a new rental transaction record for customer. The system assigns an account number to the customer and prints the bar code. Go to initial state.

Output Printing of the bar code. Customer can rent tapes.

Step-1 & Step 2 In this statement, we identify rent tape, insert customer information (based on Rule-1B) and print bar code computations. The other two computations are extracted based on the POS tagger. However, we can actually exclude the rent tape computation here as the main task is adding a new customer record and not the rent tape process. Both insert customer information and print bar code can be extracted as DT. We identify an ordering of computations, thus it is mapped to a PIPE.

Step-3 The partial architecture for R15 can be modelled as in Fig. C.58(a).

Step-4 The incremental system architecture can be designed as in Fig. C.58(b).

Issue and discussion Not applicable.
Requirement-16

Description Before a new tape can be rented all necessary information must be entered. Then a bar code is printed and the clerk has to glue it on the video.

Input A video tape is characterised by the following attributes: video name, rental price and tape ID.

Processing Create a video inventory record for tape.

Output Video inventory record produced. Tape can be rented. Print a bar code for the tape.

Step-1 & Step 2 We can directly extract insert (or enter) video information and print bar code computations. Both computations can be grouped in a single component i.e. the IV component. We identify a sequence of computations, however, since they are internal within the same component, we do not consider any composition connector.

Step-3 The partial architecture for R16 can be modelled as shown in Fig. C.59(a).

![Diagram](image)

(a) R16 (b) Incremented architecture

Fig. C.59: R16

Step-4 The incremental system architecture can be designed as in Fig. C.59(b).

Issue and discussion Not applicable.

Requirement-17

Description The clerk can change the data either of a customer or a video.

Input Clerk enters new data of either a video or a customer.

Processing Updating the data in the video inventory file.
Output  Display if the data was changed.

Step-1 & Step 2  We can extract change customer data or change video data. However, both of these computations have already been identified and modelled in R1. We identify a selection of computations based on the control term ‘either’.

Step-3  The partial architecture for R17 can be modelled as in Fig. C.60(a).

![Diagram](https://via.placeholder.com/150)

(a) R17  (b) Incremented architecture

Fig. C.60: R17

Step-4  The part of the incremented architecture for R17 is shown in Fig. C.60(b).

Issue and discussion  Not applicable.

Requirement-18

Description  Only managers can delete customers or video.

Input  Manager enters account number of video or customer.

Processing  Deleting video or customer.

Output  Display if the data was deleted.

Step-1 & Step 2  Looking at the statement, based on the POS tagger extraction rule, we identify delete customer, delete video and display computations. Apart from that, based on a descriptive expression that is “only managers”, we conclude that a computation to authenticate the manager’s status must be provided. The result of the validation status will be passed to the delete customer and delete video computations. Hence, we map this flow as a PIPE because data is passed from AUT component to the DC and DV components.
Step-3 The partial architecture for R18 can be modelled as in Fig. C.61(a).
Step-4 The incremental system architecture is shown in Fig. C.61(b).

Issue and discussion Not applicable.

Requirement-19

Description The manager can print daily reports or some statistics.

Input Manager selects what kind of information he wants to have. He can choose from the following list:

1. daily report;
2. lists of customers registered during some time period;
3. lists of customers marked bad credit;
4. lists of customers with overdue items;
5. lists of tapes by status;
6. lists of tapes not rented for a certain number of days;
7. number of rentals (by copy, title, type) over a certain time period;
8. number of days rented (by month, year, copy, and title);
9. customer rental histories.

Processing Collecting all data for the requested information and printing of it.

Output Printed data.
Step-1 & Step 2 We can directly extract computations to provide the required lists. For instance, for the first list, in order to generate a daily report, we need to provide a computation that provides such a list. For this example, we identify the computation `dailyReport`. This computation has to link with the VIR and RTR database elements to extract the record for daily report. The same case is treated to all the rest of the lists.

Step-3 The partial architecture for R19 is modelled in Fig. C.62(a).

![Fig. C.62: R19](image)

Step-4 This architecture is considered as the final architecture before the finalisation step (see Fig. C.63).

![Fig. C.63: Incremented Architecture for R19](image)

Issue and discussion Not applicable.

C.2.3 Architecture Refactoring of the VSS

Fig. C.64 shows the final VSS architecture as a result of the incremental composition. However, we identified patterns, which are labelled as AR-1 and AR-2 in Fig. C.65 that can be further simplified. These two occurring patterns can be combined into a single design instead of two. In order to preserve the incremented behaviours, when combining both the patterns (consisting of pipes and guards)
the display computation for deleting option needs to be re-arranged. Nonetheless, this change does not alter the behaviour that is expected to be preserved. This change of structure can be shown in Fig. C.66. The following step is to fix all the composition points and finalise them. See Fig. C.67.
Fig. C.64: VSS Architecture Before Refactoring

Fig. C.65: Label for VSS Architecture Refactoring
C.2. THE VIDEO STORE SYSTEM (VSS)

Fig. C.66: Refactored VSS Architecture

Fig. C.67: Finalised VSS Architecture
C.3 The Word Count (WC)

C.3.1 Introduction
This Word Count example is drawn from [Sch06, p. 434]. To simplify and ensuring consistency of the documentation, each step will be labelled as Step-1 to Step-5 accordingly. To recall:

**Step-1** Extracting elements of component-based systems from requirements. This step is performed with the help of the Extractor tool. For brevity, the screen captures are not documented.

**Step-2** Mapping of the extracted elements to X-MAN elements;

**Step-3** Creating a partial architecture;

**Step-4** Composing the partial architecture with the existing system architecture;

**Step-5** Finalising the system architecture.
Note that Step-5 is only applicable after the final requirement has been executed. An item for any issue and discussion is also added.

C.3.2 Requirements for Word Count

1. The system shall take a file name as an input.

2. Each time a file is given it must be validated. If the file name is invalid, an error message will be displayed.

3. Once validated, the words in the file will be formatted according to the type of input file.

4. The formatted file will then be counted.

5. The result of the calculation that contains the number of words in the file will be displayed to the user.

C.3.3 Incremental Execution of the Word Count

**Requirement-1**

1. The system shall take a file name as an input.

**Step-1 & Step 2** No computation or control identified here. Although we can extract take as to imply computation based on the user interaction (see HR-1B), nonetheless, such a task can be handled by a GUI process. Hence, we do not specify this verb as a computation. For data, the file name is required.
Step-3 Hence, no partial architecture for this requirement.

Step-4 Not applicable.

Issue and discussion Not applicable.

Requirement-2

2. Each time a file is given it must be validated. If the file name is invalid, an error message will be displayed.

Step-1 & Step 2 For this requirement, we identify validate and display as data transformations. From the conjunction ‘if’, we extract control and as for data, we identify ‘file name’ and ‘invalid’ as the file status. These two DTs are mapped to their corresponding computations that are validate-
File and displayError. These computations are assigned to components that provide the corresponding services. For control, the conjunction ‘if’ can either be mapped to a SEL or a guard.

Although we map the conjunction ‘if’ from the requirement statement to a guard, as the extracted computations are provided by different components, hence, we need a composition connector to compose them. We assume the execution order follows the way the requirement is written i.e. the file must be validated prior to the display computation.

Step-3 The partial architecture for R2 is as shown in Fig. C.68.

Step-4 Since this is the first partial architecture, it is considered as the system diagram.

Issue and discussion Consider other generic exogeneous connector (EC) to be used in this scenario.

1. A SEL can be used whenever a selection between two or more components has to be made. In such a case, it might be applicable to
select between these two components. However, in this particular requirement, we do not want to execute only one of the components. By adopting a SEL, we are restricted to choosing the execution between only one of the components at a time.

2. A PIPE is a generic case of a SEQ where data can be piped into the connecting components. In this case, data representing the validation status will be passed to the latter component to be evaluated by a guard. The guard here comprises a boolean condition as the validation status. Hence, it is appropriate to apply the PIPE with a guard here.

Requirement-3

3. Once validated, the words in the file will be formatted according to the type of input file.

Step-1 & Step 2 Here, we identify validate file and format file from the data transformation (DT) category. These two DTs are mapped to their corresponding computations that are validateFile and format. These computations are assigned to components that provide the respective services. There is no control that can be explicitly identified from the requirement. However, since we identify two computations provided by different components, a composition connector is required to compose them.

Step-3 The partial architecture for R3 is as shown in Fig. C.69(a).

![Image](image_url)

(a)R3    (b)Incremented architecture

Fig. C.69: Partial and Incremental Architecture for R3

Step-4 The incremented system architecture is shown in Fig. C.69(b).

Issue and discussion Not applicable.

Requirement-4

4. The formatted file will then be counted.

Step-1 & Step 2 We extract count and format from DT category. These two DTs are mapped to their corresponding computations that are countWord
and `formatFile`. These computations are assigned to their components that provide the services. The preposition ‘then’ implies ordering. Hence, we can map the control to a PIPE or a SEL. As the result of the formatted file needs to be passed to the CW component to be counted, we choose a PIPE to map the preposition instead of a SEL. For data, we identify the file and imply the result of the counted word.

**Step-3** The partial architecture for R4 is as shown in Fig. C.70(a).

![Diagram](image)

(a)R4  (b)Incremented architecture  (c)Updated architecture

Fig. C.70: Partial, Incremented and Updated Architecture for R4

**Step-4** The incremented system architecture is shown in Fig. C.70(b).

**Issue and discussion** However, once the CW component is composed with the architecture, we need to add a guard to check that the counted file is validated. Otherwise, if the file status is invalid, the file will still be passed to the CW component, in which is misleading. Fig. C.70(c) models this addition.

**Requirement-5**

5. The result of the calculation that contains the number of words in the file will be displayed to the user.

**Step-1 & Step 2** We extract `count` and `display` from DT category. These two DTs are mapped to their corresponding computations that are `countWord` and `display`. These computations are assigned to their components that provide the services, namely the CW and DIS components. There is no explicit control that can be extracted. We assume the default ordering. For data, we identify the calculation result.

**Step-3** The partial architecture for R5 is shown in Fig. C.71(a).

**Step-4** The incremented system architecture is shown in Fig. C.71(b). So far, this is considered the final architecture before the refactoring task takes place.
Issue and discussion As the CW component is already being modelled to the current system architecture, we just need to compose the DIS component. Again, a guard to check that the counted file is validated is required. Fig. C.71(c) models this addition.

C.3.4 Architecture Refactoring of the WC

Based on the initial system architecture before refactoring, we look for any pattern that can be simplified. Here, we identify the use of mutually exclusive guards for checking the file status, either valid or invalid. These guards can be simplified with a SEL. However, to derive a meaningful composition, the execution when the file status is valid, is composed using a PIPE. If we simply adopt a SEL, the composition will not be accurately modelled as there are only two options to be branched (i.e. valid and invalid). Hence, the resulting architecture is re-structured using a new PIPE as shown in Fig. C.72.

C.3.5 Finalisation step

The following step is to finalise all the open composition points. This is depicted in Fig. C.73.
C.4 Summary

This example has demonstrated the application of the heuristics and design principles defined in the incremental approach. Starting by extracting elements of component-based systems from NLR, these elements are mapped to their corresponding architectural constructs defined in the X-MAN model; whilst constructing the architecture incrementally. After refactoring, the final system architecture is eventually devised.

Fig. C.73: Finalisation Step.