Scenario-driven Development and Runtime Evolution of Context-aware Adaptive Software Systems

Mahmoud Hussein

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, Australia

2013
Abstract

There is an increasing demand for software systems that have the ability to be adapted at runtime in response to changes in their environments and requirements. The changes can either be anticipated at the development time, or become known only when the systems in operation (i.e. unanticipated changes). Thus, such context-aware adaptive systems need to be developed with the adaptability in mind to cope with the anticipated changes, and to evolve in response to the unanticipated changes.

To enable the development and runtime evolution of a context-aware adaptive system, a set of challenges need to be tackled. First, the system’s functional and adaptation requirements and the context information required by the system for functional and adaptation use need to be explicitly specified in an easy-to-understand form. Second, the adaptation requirements of the system need to be checked for consistency to ensure that consistent adaptation actions are only applied to the system at runtime, and for validity to ensure that the application of these actions leads to valid system variants. Third, the system’s executable model needs to have the ability to be adapted at runtime in response to anticipated and unanticipated changes. Fourth, to maintain a causal connection between the system’s executable model and its requirements and to ease the task of designing the system, the executable (runtime) model of the system needs to be derived from its requirements automatically.

In this thesis, we present a novel approach to assist the software engineer in the development and runtime evolution of context-aware adaptive systems. Our approach is scenario-driven, where the requirements for a context-aware adaptive system are specified as a set of scenarios. These scenarios are then automatically validated to ensure their consistency and validity. In addition, the system scenarios are used to derive the system’s executable model for deployment. Furthermore, to cope with the unanticipated changes, the system scenarios are changed and their changes are then automatically reflected to the system while it is in operation.

The approach makes four main contributions. First, it specifies the requirements for a context-aware adaptive software system as two sets of scenarios, i.e., functional and adaptation scenarios. These scenarios also identify contexts and their functional and adaptation use in the system. Second, the approach supports the automatic validation of the large number of system variants introduced by its high variability. It checks the consistency of the system’s adaptation requirements, and identifies the system variants which are then generated and checked relative to system properties (that should hold at runtime) to ensure their validity. Third, we introduce an organization-based meta-model and adopt the models@runtime concept for keeping a context-
aware adaptive system’s model (following the meta-model) alive at runtime. Thus, the system’s executable (runtime) model can be easily adapted in response to anticipated and unanticipated changes. Fourth, to ease the task of designing the executable model of a system, we introduce a technique to derive the system’s executable model from its requirements. In addition, to cope with unanticipated changes, we introduce a set of evolution patterns. These patterns are used to derive the changes to the system’s runtime model from its changed scenarios (requirements).

To demonstrate the applicability of our approach, we have used it to develop and evolve two case studies: a travel guide system and an electronic exam system. In addition, a feature-based analysis is performed to evaluate to what extent the approach meets the requirements that need to be considered in developing and evolving context-aware adaptive systems. Furthermore, a quantitative evaluation of the approach is carried out for: (1) assessing the reduction in the engineering effort gained when our approach is used for developing and evolving context-aware adaptive systems; (2) measuring the runtime cost of adding the adaptability feature to a software system; (3) quantifying the computational complexity for the algorithms (techniques) introduced in the approach.

Overall, our novel scenario-driven approach assists the software engineer in specifying and validating the requirements of a context-aware adaptive system, and in designing, realizing, and evolving the system. In particular, it enables runtime changes to the software system in response to anticipated and unanticipated changes, and reduces the engineering effort needed to develop and evolve the system.
Acknowledgments

I would like to express my sincere thanks to my supervisor Prof. Jun Han for his generous help during my candidature and for creating a research environment that has been very inspiring. Thank you very much for the abundant hours spent discussing the research topic, reviewing papers and drafts, and providing valuable advices. My warm thanks also to my co-supervisors Dr. Alan Colman and Dr. Jian Yu for their continuous support, advices, and feedback.

Many thanks to the members of the Context-aware Adaptive Architecture research group (Minh Tran, Adeel Talib, Malinda Kapuruge, Ayman Amin, Ashad Kabir, Tuan Nguyen, Tharindu Patikirikoral, Indika Kumara, and A.S.M Kayes) for their discussions and feedback that positively influenced the outcome of the thesis.

This thesis would not have been possible without the scholarships provided by Swinburne University of Technology and the Australian’s Cooperative Research Centre for Advanced Automotive Technology (AutoCRC).

My deepest gratitude goes to my family for their love, support, and encouragement during the period of conducting the research and writing up this thesis.

I must also thank my friends Khalid Mansour, Mohammed Almorsey, Mohammed Abd-Elkawy, and Marawan Ahmed for all the support during my stay in Melbourne.

Finally, to all of you that deserve thanks but are not mentioned here, Thanks!
Declaration

This is to certify that,

- This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the examinable outcome; and

- To the best of my knowledge, the thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis; and

- Where the work is based on joint research or publications, the thesis discloses the relative contributions of the respective workers or authors.

Mahmoud Hussein

October, 2013

Melbourne, Australia.
# Table of Contents

Abstract .............................................................................................................................. i
Acknowledgments ............................................................................................................ iii
Declaration ....................................................................................................................... iv
List of Figures .................................................................................................................. ix
List of Tables .................................................................................................................. xii
List of Listings ............................................................................................................... xiii
List of Publications ......................................................................................................... xv

Part I: Context-aware Adaptive Systems .......................................................................... 1

Chapter 1 : Introduction .................................................................................................... 2
  1.1 Context-aware Adaptive Software Systems ............................................................ 2
    1.1.1 Context-aware and Adaptive Systems ............................................................. 3
    1.1.2 Development and Evolution of Context-aware Adaptive Systems ............... 5
  1.2 Research Objectives ............................................................................................... 7
  1.3 The Approach Overview ......................................................................................... 8
  1.4 Research Contributions ......................................................................................... 12
  1.5 Thesis Outline ........................................................................................................... 12

Chapter 2 : Motivational Scenario .................................................................................. 16
  2.1 The Travel Guide Software System ........................................................................ 16
    2.1.1 Composing a Context-aware System ............................................................ 16
    2.1.2 Runtime Adaptability to Cope with Anticipated Changes ............................ 17
    2.1.3 Runtime Evolution in Response to Unanticipated Changes ....................... 18
  2.2 Challenges for Engineering Context-aware Adaptive Systems ............................ 18
    2.2.1 Specifying the System Requirements ............................................................ 18
    2.2.2 Validating the System Requirements ............................................................. 19
    2.2.3 Designing and Realizing the System ............................................................. 20
      2.2.3.1 Designing Context-aware Adaptive Systems ......................................... 20
      2.2.3.2 Realizing Context-aware Adaptive Systems ........................................... 22
    2.2.4 The System’s Runtime Evolution ................................................................. 22
    2.3 Summary ............................................................................................................... 23

Chapter 3 : Literature Review ........................................................................................ 25
  3.1 Classification of Approaches for Developing and Evolving Context-aware Systems ... 25
  3.2 Specifying the System Requirements ..................................................................... 27
    3.2.1 Approaches based on General Goal Models ................................................. 27
    3.2.2 i*/Tropos-based Approaches ....................................................................... 31
    3.2.3 Problem Frames-based Approaches ............................................................. 34
    3.2.4 Summary ........................................................................................................... 35
  3.3 Validating the System Requirements ...................................................................... 36

6.1 Specifying and Validating the System Requirements’ Changes ........................................... 131
   6.1.1 Specifying the Requirements Changes ................................................................. 131
      6.1.1.1 Changes to the Functional Requirements ........................................................ 132
      6.1.1.2 Changes to the Adaptation Requirements .......................................................... 135
   6.1.2 Validating the Requirements Changes ................................................................. 136

6.2 Synthesizing the Changes to the System’s Design Model .............................................. 139
   6.2.1 The Evolution Patterns ............................................................................................. 140
      6.2.1.1 A Template for the Evolution (Change) Patterns ............................................... 141
      6.2.1.2 Evolution Patterns to Cope with Unanticipated Context Changes ................... 142
      6.2.1.3 Evolution Patterns to Accommodate Changes in a System’s Functionality ...... 146
      6.2.1.4 Evolution Patterns to Incorporate Changes to the Adaptive Behaviour ............ 150
   6.2.2 An Algorithm to Synthesize the Evolved Design Model of the System ............... 152

6.3 Realizing Runtime Changes to the System ..................................................................... 154

6.4 Summary ......................................................................................................................... 159

Chapter 7 : Tool Support ............................................................................................... 160

7.1 An Overview of the Toolset ............................................................................................ 160

7.2 System Requirements Specification ................................................................................ 162

7.3 Requirements Validation ................................................................................................ 165

7.4 System Design and Realization ..................................................................................... 167

7.5 Runtime System Evolution ............................................................................................. 171

7.6 Summary ......................................................................................................................... 172

Part III: Evaluation ........................................................................................................ 173

Chapter 8 : Case Studies ............................................................................................... 174

8.1 The Travel Guide Software System ................................................................................ 174

8.2 An Electronic-Exam Management System .................................................................... 178
   8.2.1 Specifying the System Requirements .................................................................... 178
   8.2.2 Validating the System Requirements ................................................................... 184
   8.2.3 Synthesizing the Design Model of the E-exam System ....................................... 184
   8.2.4 Evolving the E-exam Management System ......................................................... 188
      8.2.4.1 Evolving the System Scenarios ......................................................................... 188
      8.2.4.2 Validating the Requirements’ Changes ............................................................... 192
      8.2.4.3 Updating the System Design Model ................................................................. 193
      8.2.4.4 Realizing the Changes in the System’s Design Model ...................................... 195

8.3 Lessons Learnt from the Case Studies ............................................................................ 196

8.4 Summary ......................................................................................................................... 199

Chapter 9 : Evaluation ................................................................................................... 201

9.1 Feature-based Evaluation of the Approach ..................................................................... 202
   9.1.1 The System Requirements Specification ............................................................... 202
   9.1.2 Validating the System Requirements ................................................................... 203
   9.1.3 Designing and Realizing the System .................................................................... 204
   9.1.4 The System Evolution ......................................................................................... 208

9.2 Effectiveness of the Approach Notations ..................................................................... 210
9.2.1 The Physics of Notations Theory ................................................................. 210
9.2.2 Evaluating the Notation for Specifying the System Requirements ........... 212
9.2.3 Evaluating the Notation for Designing the System .................................... 214
9.2.4 Summary ..................................................................................................... 216
9.3 Reduction in Engineering Effort ................................................................. 217
  9.3.1 Sources of Effort Reduction ....................................................................... 217
    9.3.1.1 Specifying the System Requirements and Properties ....................... 217
    9.3.1.2 Automatic Validation of the System Requirements ......................... 218
    9.3.1.3 Synthesis of the System’s Desing Model from its Requirements ........ 218
    9.3.1.4 Generating the System Implementation from the Design Model ...... 219
    9.3.1.5 Automatic Realization of the System’s Runtime Changes ............... 219
    9.3.1.6 Summary .......................................................................................... 220
  9.3.2 Estimating the Effort Reduction ............................................................... 221
    9.3.2.1 Background: COCOMO II ............................................................... 221
    9.3.2.2 Effort Reduction in Developing and Evolving the Travel Guide System 223
    9.3.2.3 Effort Reduction in Developing and Evolving the Electronic Exam System 228
9.4 Performance Evaluation ............................................................................. 232
  9.4.1 The Time Required for Adapting a System at Runtime ......................... 232
  9.4.2 The Time Complexity of the Algorithms ............................................... 234
9.5 Summary ..................................................................................................... 235

Chapter 10: Conclusion .................................................................................. 237
  10.1 Contributions ............................................................................................ 237
  10.2 Future Work ............................................................................................ 240

Appendices ...................................................................................................... 245
  Appendix A: The Travel Guide System’s Scenarios, Properties, and ROAD Model 246
  Appendix B: The Electronic Exam System’s Scenarios and Properties .......... 266
  Appendix C: The Effort Multipliers and Scale Factors of COCOMO II .......... 281
  Appendix D: Reduction in the Development Time and Cost ...................... 289
List of Figures

Figure 1-1: Abstract layered architectures for context-aware and adaptive software systems [4, 15] ........ 3
Figure 1-2: The basic elements of a context-aware adaptive software system and their relationships......... 5
Figure 1-3: A process for developing and evolving a context-aware adaptive software system ............. 9
Figure 2-1: Adapting the travel guide system to include the attractions finder service ...................... 17
Figure 3-1: A classification of existing approaches based on the engineering tasks............................ 26
Figure 3-2: A goal model that defines different alternatives of the meeting scheduling system (Cf. [80]) 28
Figure 3-3: Example adapt cases in the znn.com system (Cf. [84]) .................................................... 30
Figure 3-4: Graphical adapt cases in the awareness requirements (Cf. [19])...................................... 31
Figure 3-5: Requirements and context models for the travel guide system (Cf. [90]) ......................... 33
Figure 3-6: Graphical representations of the formal notations to specifying a system’s properties ......... 38
Figure 3-7: A formal model for the mine pump system’s requirements (Cf. [30]) .............................. 41
Figure 3-8: The main elements of the Rainbow framework (Cf. [16]).............................................. 45
Figure 3-9: Designing an adaptive system (application) using the CASA framework (Cf. [127]) ......... 48
Figure 3-10: Developing and updating a software system using the Ginseng approach (Cf. [68]) ......... 59
Figure 3-11: Developing and evolving context-aware adaptive software systems ............................. 63
Figure 4-1: Runtime adaptation in context-aware software systems .................................................... 68
Figure 4-2: Specifying a system’s variants and the switching between two of them in our approach 68
Figure 4-3: A meta-model for the extended UML sequence diagram to specify the functional scenarios. 70
Figure 4-4: An example scenario that shows the user interactions with the travel guide provider........... 70
Figure 4-5: Sequence of interactions to suggest a number of attractions to the user ............................ 73
Figure 4-6: A functional scenario for planning a route in the travel guide system ................................. 73
Figure 4-7: A meta-model for the extended UML sequence diagram to capture the adaptation scenarios 74
Figure 4-8: An adaptation scenario to cope with changes in the user selected features (services) ............. 75
Figure 4-9: An adaptation scenario to cope with changes in the traffic information availability ........... 76
Figure 4-10: Example variants generated from the scenario “FS3” by the adaptation scenario “AS2” ....... 77
Figure 4-11: A meta-model for the extended UML sequence diagram to capture the system properties .. 78
Figure 4-12: Example properties that need to be preserved while the travel guide system in operation.... 79
Figure 4-13: A process for validating a system’s functional variants.................................................... 80
Figure 4-14: The different variants of the scenario “FS1” generated by the adaptation scenario “AS1” .... 84
Figure 4-15: Two variants of the functional scenario “FS1” ............................................................... 85
Figure 4-16: The variants generated by the adaptation scenario “AS1” and the switching between them 86
Figure 4-17: Example scripts and variants generated from the travel guide system’s scenarios .............. 86
Figure 4-18: Example switching between three variants of the travel guide software system .............. 87
Figure 4-19: Example variants of the scenario “FS1” modelled as Petri nets ....................................... 88
Figure 5-1: A process for designing and realizing context-aware adaptive systems ............................ 91
Figure 5-2: Example functional and management composites of a context-aware adaptive system ....... 92
Figure 5-3: A meta-model for a context-aware adaptive system’s functional composite .................... 95
Figure 8-7: A behaviour process corresponding to the functional scenario “FS1” ................................... 186
Figure 8-8: The structure of the e-exam system’s management composite.............................................. 187
Figure 8-9: Changes to the e-exam system’s functional scenarios “FS1” and “FS7” ............................... 189
Figure 8-10: A scenario that shows the tasks to be performed by a student to take an exam practice .... 190
Figure 8-11: Changes to the adaptation scenario “AS1” ............................................................................ 191
Figure 8-12: Example properties that are added to the e-exam system ..................................................... 192
Figure 8-13: The evolved functional composite of the e-exam system ..................................................... 193
Figure 8-14: The evolved behaviour process for a student’s interactions with the e-exam system .......... 194
Figure 9-1: The estimated effort for developing the travel guide system using the Monte Carlo simulation .......................................................................................................................... 226
Figure 9-2: The estimated effort for evolving the travel guide system using the Monte Carlo simulation ........................................................................................................................................... 227
Figure 9-3: The estimated effort for developing the e-exam system using the Monte Carlo simulation .. 230
Figure 9-4: The estimated effort for evolving the e-exam system using the Monte Carlo simulation ..... 231
Figure 9-5: The adaptation rules’ loading and execution times ................................................................. 233
List of Tables

Table 3-1: A summary of the proposed approaches to specify a system’s requirements.......................... 35
Table 3-2: A summary of approaches to specify a system’s properties and to validate its requirements.. 43
Table 3-3: A summary of approaches to design and realize context-aware adaptive systems................. 54
Table 3-4: A summary of the approaches for evolving context-aware adaptive systems ..................... 61
Table 5-1: The mappings between the system scenarios and its design model....................................... 107
Table 5-2: The mappings between the management messages and the adaptation actions .................... 116
Table 5-3: Mappings between a system model designed in our approach and the ROAD model .......... 121
Table 6-1: Evolution (change) patterns to cope with the unanticipated changes .................................. 140
Table 6-2: The mappings between the scenarios’ changes and the changes to the design model .......... 152
Table 8-1: Summary of the two case studies ......................................................................................... 177
Table 8-2: The e-exam system’s functional scenarios ............................................................................ 180
Table 8-3: The e-exam system’s adaptation scenarios .......................................................................... 182
Table 8-4: Summary of the e-exam system’s properties ..................................................................... 183
Table 8-5: The changes to the functional scenarios of the e-exam system ........................................... 189
Table 8-6: Incorporating changes to the e-exam system’s adaptation scenarios ................................... 191
Table 8-7: Summary of the e-exam system’s properties ..................................................................... 192
Table 9-1: Comparing our approach with the approaches that support the requirements specification ... 203
Table 9-2: Comparison between our approach and approaches to validating the system requirements ... 204
Table 9-3: Comparing our approach with approaches that support the system design and realization ... 207
Table 9-4: Comparison between our approach and the approaches that enable the system evolution .... 209
Table 9-5: Summary of evaluating the notations to specifying the system requirements and design ..... 216
Table 9-6: Summary of the engineering tasks and how they are performed in our approach ............... 220
Table 9-7: Effort estimation for developing and evolving the travel guide software system ................. 224
Table 9-8: Summary of the average effort reduction in developing and evolving the travel guide system ................................................................................................................................. 228
Table 9-9: Effort estimation for the development and evolution of the e-exam system ......................... 229
Table 9-10: The average effort, time, and cost for developing and evolving the e-exam system ........... 231
Table 9-11: The time (in milliseconds) required to apply the adaptation actions ................................. 233
List of Listings

Listing 4-1: An algorithm (pseudocode) to generate and validate the system variants ........................................ 82
Listing 5-1: A high level description for the functional composite of the travel guide system ......................... 96
Listing 5-2: The functional contract “FC2” ........................................................................................................ 97
Listing 5-3: The contextual contract “CC1” ........................................................................................................ 98
Listing 5-4: The tasks of the route planner and the traffic information roles ..................................................... 99
Listing 5-5: The description of the “P3: PlanRoute” process .............................................................................. 100
Listing 5-6: A high level description of the management composite for the travel guide system ............... 103
Listing 5-7: Parts of the description for the travel guide system’s organizer player ....................................... 105
Listing 5-8: Part of the management contract “MC1” ....................................................................................... 105
Listing 5-9: An algorithm (pseudocode) to synthesize a composite’s roles from a set of scenarios ........ 108
Listing 5-10: An algorithm to synthesize a system’s functional contracts ..................................................... 109
Listing 5-11: An algorithm to synthesize a system’s behaviour processes (Part 1) ........................................ 111
Listing 5-12: An algorithm to synthesize a system’s behaviour processes (Part 2) ...................................... 112
Listing 5-13: Synthesis of conversation clauses and temporal constraints from the scenarios .................. 114
Listing 5-14: An algorithm to synthesize a system’s adaptation rules and management contract .......... 115
Listing 5-15: Part of the XML description for the travel guide system following the ROAD approach .. 118
Listing 5-16: An example contract in the travel guide system ........................................................................ 119
Listing 5-17: An XML description of a process in the travel guide system by the ROAD approach .... 120
Listing 5-18: An example rule file for the contract “FC2” ............................................................................. 121
Listing 5-19: An algorithm to generate an executable ROAD model from a system’s design model .... 122
Listing 5-20: An example transformed contextual contract in the travel guide system ............................. 123
Listing 5-21: A Java code corresponding to the functional player “RoutePlanner1” ................................ 124
Listing 5-22: An example adaptation rule in the Drools format ..................................................................... 125
Listing 5-23: Deciding the adaptation actions in response to context changes ............................................ 126
Listing 5-24: Acting the adaptation actions that are generated in response to context changes ................. 127
Listing 6-1: An algorithm for generating the new adaptation scripts ............................................................. 138
Listing 6-2: An algorithm for identifying and generating the new system variants ...................................... 138
Listing 6-3: An evolution pattern to add new functionality to a system’s design model ............................ 141
Listing 6-4: An evolution pattern to add a new context attribute to a software system ............................. 143
Listing 6-5: The modified contextual contract “CC2” .................................................................................. 144
Listing 6-6: An evolution pattern to add a new context entity to a software system .................................. 144
Listing 6-7: The new contextual contract “CC4” ......................................................................................... 145
Listing 6-8: An evolution pattern for changing the context quality .............................................................. 145
Listing 6-9: The modified contextual contract “CC1” .................................................................................. 146
Listing 6-10: An evolution pattern to remove a context attribute or a context entity ............................... 146
Listing 6-11: The description of the “P4: FindRestaurant” process ................................................................ 147
Listing 6-12: An evolution pattern to cope with changes in the system’s existing functionality ............ 148
Listing 6-13: Part of the modified functional contract “FC2” .............................................................. 148
Listing 6-14: An evolution pattern to cope with changes in the system’s non-functional requirements .. 149
Listing 6-15: An evolution pattern to add a new adaptive behaviour to the system ............................. 150
Listing 6-16: Parts of the description for the modified travel guide system’s organizer player .............. 151
Listing 6-17: Part of the modified management contract “MC1” ............................................................ 151
Listing 6-18: An evolution pattern to change the system’s adaptive behaviour ................................. 152
Listing 6-19: An algorithm to synthesize an updated system design from the changed scenarios ......... 153
Listing 6-20: An algorithm (pseudocode) to compute differences between two system models .......... 155
Listing 6-21: An algorithm (pseudocode) to identify changes in a system’s functional contracts ......... 156
Listing 6-22: Part of the script to evolve the functional composite of the travel guide system ............. 157
Listing 6-23: Creating a Java method to apply an evolution script to the running system .................. 158
Listing 7-1: Output statements during the execution of the algorithm to generate the system variants ... 165
Listing 8-1: A high level description for the functional composite of the e-exam system .................... 185
Listing 8-2: Part of the functional contract “FC6” .................................................................................. 186
Listing 8-3: The description of the process “P1: UsingSystemByStudent” ........................................... 186
Listing 8-4: A high level description of the management composite for the e-exam system ............... 187
Listing 8-5: Part of the management contract “MC1” .......................................................................... 188
Listing 8-6: Part of the description for the e-exam system’s organizer player ................................. 188
Listing 8-7: Part of the modified functional contract “FC6” ............................................................... 194
Listing 8-8: The description of the evolved process “P1: UsingSystemByStudent” ............................ 195
Listing 8-9: Parts of the description for the changed organizer player of the e-exam system ............ 195
Listing 8-10: Part of the script for evolving the functional composite of the e-exam system ............ 196
List of Publications

The following papers, which I am the primary author, have been accepted and published during my candidature. The thesis is largely based on these papers.


Part I: Context-aware Adaptive Systems
Chapter 1: Introduction

There is an increasing demand for software systems that are able to adapt their behaviours at runtime in response to changes in their environments and requirements [1]. These changes are either anticipated at the development time [2], or they become only known while the systems are in operation (i.e. unanticipated changes) [3]. First, at the development time, changes to a system’s environment can be partially anticipated. Thus, the system needs to be developed with the ability to adapt its behaviour in response to such anticipated changes without explicit user intervention to bring better usability and effectiveness [2, 4]. Second, while a software system is in operation, it needs to evolve in response to unanticipated changes [3, 5]. The changes can be in its environment or requirements where: (a) the system is deployed into an environment that is not totally anticipated at the development time; (b) the provider wants to enhance the system with a new feature or the users may need a new functionality. This thesis addresses the problem of how to develop software systems that have the ability to adapt themselves in response to anticipated changes in their environments, and enable their runtime evolution in response to the unanticipated changes. We call such systems “context-aware adaptive software systems”.

In this chapter, we introduce the research context, the research objective, and outline the approach to achieve the research objective. In Section 1.1, we present the research context by giving an overview of context-aware adaptive software systems and the challenges that face the development and evolution of these systems. The research objective is described in Section 1.2. In Section 1.3, we give an overview of our approach. The original contributions of the thesis are listed in Section 1.4. Finally, the thesis structure is given in Section 1.5.

1.1 Context-aware Adaptive Software Systems

Modern software systems need to be developed with adaptability in mind to cope with runtime changes in their environments and requirements [4-6]. These changes can be anticipated at the development time, or they become only known at runtime (i.e. unanticipated changes). Thus, such software systems need to have the ability to adapt themselves at runtime in response to the anticipated changes [7], and to evolve while they are in operation to take the unanticipated changes into account [8].

In recent years, a number of approaches have been introduced to support the development and evolution of context-aware adaptive software systems. In the following, we describe what context-aware adaptive systems are, and the challenges that still face the development and runtime evolution of such software systems.
1.1.1 Context-aware and Adaptive Systems

Research into software systems that have the ability to adapt themselves in response to changes in their environments and requirements has been conducted primarily by researchers from two communities with two different perspectives: context-aware systems [2, 4] and adaptive systems [3, 5]. In the following, we describe these two perspectives and discuss how to consider them in a holistic manner to form what we call “context-aware adaptive software systems”.

First, context-aware software systems are “systems that are able to adapt their operations in response to context changes without explicit user intervention” [4, 9]. Their aim is to increase the usability and effectiveness of the software systems by taking the environmental context into account. Context is defined as “any information that characterizes the situation of environment entities (e.g. persons, places, or objects) that are relevant to the interaction between a system and its end users” [9]. To develop context-aware software systems, Matthias et al. introduced a layered architecture for such systems (shown in Figure 1-1.A) [4]. This architecture contains elements that are common to most of existing context-aware systems in five layers [10-14]. The first layer (i.e. the bottom layer) is the hardware and/or software sensors that detect changes in the system environment, and then the context changes are retrieved from the sensors by the second layer. At the third layer, the collected context data is interpreted to infer high level context information (if any). Layer four organizes the context information and offering them to the functional system. Finally, layer five represents the system’s core functionality and the actual reaction(s) to the context changes.

![Figure 1-1: Abstract layered architectures for context-aware and adaptive software systems [4, 15]](image)

Second, adaptive software systems are “systems that are able to modify their own structures and/or behaviours in response to changes in their requirements and/or operating environments” [16-17]. Kramer and Magee introduced a layered architecture for such adaptive systems [15].
The architecture has three layers that need to be considered in developing an adaptive system as shown in Figure 1-1.B. Layer one (i.e. component control) contains a number of interconnected components that represent the system’s core functionality. This layer also includes facilities to report components’ status to the second layer, and to support component(s) creation, deletion and interconnection to execute actions decided by layer two. The change management layer uses a number of pre-computed plans to decide adaptation actions that need to be performed in response to changes coming from the component control layer. It also notifies the third layer of the changes that violate the system goals but the system does not have plans to cope with these changes. Layer three (i.e. goal management) has specification of the system goals, and it produces plans to maintain the system goals when one of them is violated and the existing plans are not able to cope with this violation or new system goals are introduced.

Context-aware systems are more concerned with how to model, process, and manage the context information (i.e. context-awareness). But, they are limited on how a system adapts itself in response to (unanticipated) changes in the context information. In addition, they are usually not concerned about runtime changes to the system requirements. On the other hand, adaptive systems are more about how to adapt the system in response to runtime changes in its context or requirements (i.e. adaptability) by separating the system functionality from its management, but pay less attention to how context is modeled, processed, managed, and made available to the system. However, modern software systems that are operating in dynamic environments need to consider the context-awareness and adaptability aspects in a holistic manner. First, modern systems need to take the context information into account to give better suggestions to the users (e.g. considering the traffic information when suggesting a set of routes to the driver). Second, while such software systems are in operation, they need to adapt themselves in response to context changes that are anticipated at the development time. For example, a system may customize itself to fit user needs or to fit the environment that it is working in. They also need to be adapted in response to changes in their environments or requirements that become known only at runtime (i.e. unanticipated changes). We call software systems that consider the context-awareness and adaptability aspects in a holistic manner as “context-aware adaptive software systems”.

A context-aware adaptive software system has a set of elements (i.e. environment, functional system, and system management) that are related to each other as shown in Figure 1-2. First, the environment represents entities that are outside the functional system boundary and affect its operation. Second, the functional system consists of elements that are responsible for providing the system’s core functionality. Third, the system management is responsible for adapting the
system in response to context changes. It has four main operations [18]: monitoring, analysing, deciding, and acting (see Figure 1-2). The monitoring operation is responsible for detecting changes in the system and its environment using a set of sensors and monitors. The analysing operation infers high level context information from low level context data that is collected by the monitoring operation. The deciding operation specifies adaptation actions in response to anticipated context changes. The acting operation is used for applying the actions coming from the deciding operation to the functional system using a set of actuators. In addition, there is an element that is responsible for evolving the system in response to unanticipated changes in its environment and/or requirements (i.e. the system’s runtime evolution shown in Figure 1-2).

![Figure 1-2: The basic elements of a context-aware adaptive software system and their relationships](image)

The environment (context) changes have two relationships with the system: operational and management. The operational relationship exists when the context is needed by the system to continue its operation, while the management relationship exists when the context changes cause the system’s runtime adaptation. Figure 1-2 shows these two kinds of relationships. First, the operational relationship is represented by direct interactions between the functional system and the environment where the system needs the context information to operate effectively. Second, the management relationship is captured using the four management operations (shown in Figure 1-2), where the system is adapted in response to anticipated context changes.

1.1.2 Development and Evolution of Context-aware Adaptive Systems

In recent years, a number of approaches have been introduced to assist the software engineer in performing the engineering tasks that are used for developing a context-aware adaptive system that has the ability to adapt itself in response to anticipated context changes, and for evolving the system at runtime to cope with unanticipated changes in its environment and/or requirements.
These tasks include specifying the system’s functional and adaptation requirements, validating these requirements, designing and realizing the system based on its requirement, and evolving a running instance of the system in response to unanticipated changes. However, these approaches have a number of limitations. In the following, we describe these limitations briefly and more details can be found in Chapter 3.

First, in existing approaches (e.g. [19-21]), the adaptation requirements of a context-aware adaptive system are intertwined with its functional requirements specification. In addition, these approaches do not explicitly capture the context information and its relationships with the system’s functional and adaptation requirements. Furthermore, the system requirements are captured as a set of goals. But, it is often difficult for the stakeholders to articulate their needs as a number of goals [22]. Therefore, it is difficult to use such approaches to specify the functional, contextual, and adaptation requirements of a context-aware adaptive software system, and an approach is needed to ease the task of specifying the three aspects of the requirements.

Second, to validate a context-aware adaptive system’s functional requirements using existing approaches, the system variant specifications need to be manually enumerated and specified by the software engineer (e.g. [23-24]). However, in complex systems that have high variability, the number of variants is large. In addition, existing approaches do not check the consistency of a system’s adaptation requirements to ensure that only valid actions are applied to the system at runtime (e.g. [25-26]). As such, there is a need for an approach that assists the software engineer in enumerating and specifying the large number of a software system’s variants, and in ensuring the consistency of a system’s adaptation requirements.

Third, in existing approaches (e.g. [16, 27-28]), a software system’s aspects (i.e. context, functionality, and management) and their relationships are not explicitly designed and realized. As such, these aspects and their relationships cannot be clearly captured, or easily changed at runtime in response to anticipated and unanticipated changes. In addition, to design a context-aware adaptive software system using existing approaches, the software engineer is responsible for translating the system requirements to a design model manually. Thus, the causal connection between the system’s design model and its requirements is maintained manually which is a complex task when the system is large. To tackle these challenges, an approach is needed that explicitly models a context-aware adaptive system’s aspects and their relationships, so that the aspects and their relationships can be clearly captured, and the realization of such aspects can be easily changed at runtime. Furthermore, it needs to support the automatic synthesis of the design model of the system from its requirements to ease the task of designing the system and to maintain the causal connection between the system requirements and the design model.
Fourth, to support the runtime evolution of a context-aware adaptive system in response to unanticipated changes, existing approaches allow the software engineer to change the system requirements specification and its design model in response to such unanticipated changes in an ad-hoc manner (e.g. [8, 29]). In addition, to validate the changes to the system requirements using the existing techniques (e.g. [23, 30]), the changed system variants need to be manually identified and specified by the software engineer which is a difficult task when a large number of variants need to be specified. Furthermore, a number of approaches have been introduced to realize the system’s design changes to a running system instance (e.g. [31-34]). But, using such approaches is tedious and error prone, where the changes need to be specified at a lower level of abstraction. Thus, an approach is needed to support the engineer in specifying and validating the changes to a system’s requirements, reflecting these changes to the system’s design model, and realizing the changes in the design model to the running system instance automatically.

1.2 Research Objectives

A set of engineering tasks need to be performed during the development and runtime evolution of context-aware adaptive software systems. The tasks to be performed at the development time aim at building a system that takes the context information into account, and able to adapt itself in response to anticipated context changes. These tasks include the system’s requirements specification and validation, designing the system model, and realizing the system. Following these tasks, a system is built and ready for operation. While the system is in operation, a number of tasks are performed to realize the system’s runtime evolution in response to unanticipated changes. These tasks are specifying and validating the system requirements’ changes, reflecting these changes to the system’s design model, and realizing the changes in the design model to the running system. This research aims to support the software engineer in doing the above tasks, and it can be summarized as follows:

“The thesis goal is to develop an approach that assists the software engineer in developing a software system that is able to dynamically adapt itself in response to anticipated context changes, and in evolving the software system to cope with unanticipated changes in its environment and/or requirements.”

In addition, the goal can be detailed into a number of technical research objectives:

- Provide a technique to specifying the requirements and properties of a context-aware adaptive system. This technique captures the requirements from three perspectives (i.e. functional, contextual, and adaptation requirements). The system requirements
and its properties should be captured in a form that is easily understandable by the stakeholders.

- Enable the validation of a software system’s functional and adaptation requirements by introducing a technique that enumerates and generates the system variants from the requirements specification. The variants are then transformed to formal models to enable their formal validation. It also checks the consistency of the adaptation requirements to ensure that only valid actions are applied to the system at runtime.

- Provide a meta-model to enable the design of context-aware adaptive systems, and a set of algorithms to synthesize a system’s design model from its requirements. The meta-model enables the design of the system aspects (i.e. context, functionality, and management) explicitly, so that these aspects and their relationships can be clearly captured and the realization of the aspects can be easily changed at runtime. The set of algorithms generate the system’s design model from its requirements to maintain the causal connection between the system design and its requirements.

- Introduce a set of techniques to enable a system’s runtime evolution in response to unanticipated changes in its environment and/or requirements. These techniques assist the software engineer in specifying and validating changes to the system requirements in response to the unanticipated changes, and reflecting these changes to the system design and the system runtime realization automatically.

1.3 The Approach Overview

To address the above mentioned research objectives, we introduce a scenario-driven approach to support the development and runtime evolution of context-aware adaptive software systems. In this section, we give an overview of our approach in terms of the process that needs to be followed in developing and evolving such systems. This process is divided into two stages.

The first stage is the system development (the top of Figure 1-3). In this stage, the system requirements are specified and validated. The system is designed based on its requirements, and the system design is translated to a deployable system. The second stage (the system evolution) starts when there is a running instance of the system (the bottom of Figure 1-3). During this stage, the system’s requirements specification is changed to incorporate unanticipated changes, and the changed requirements specification is validated. Then, the system design is changed in response to the unanticipated changes, and the design changes are reflected to the running system. Below, we describe these two stages in detail and our approach to support them.
Specifying the Requirements (Step 1): The success of a software system is measured by the degree to which it meets the users’ requirements, so that specifying these requirements is an important step in developing the software system. To specify a context-aware adaptive system’s requirements, two types of requirements need to be captured: functional and adaptation. The functional requirements specify the system functions and the context information that is needed by these functions to operate effectively [35]. The adaptation requirements capture the system reactions to anticipated context changes [36].

To specify a context-aware adaptive software system’s requirements, we adopt a scenario-based approach where the system requirements are specified as two sets of scenarios: functional and adaptation. The functional scenarios capture the system functionality, while the adaptation scenarios specify the system reactions to runtime context changes. In such scenarios, we specify context and its operational and management relationships with the system explicitly. The system properties that need to hold at runtime are also specified in a form similar to the scenarios. We adopt a scenario-based approach because its simplicity and intuitive graphical representation facilitate stakeholders involvement in specifying a software system’s requirements [22]. It also has a well-understood and widely accepted semantics [37].

Figure 1-3: A process for developing and evolving a context-aware adaptive software system
Validating the Requirements (Step 2): The adaptation requirements specify changes to the system functionality in response to context changes. To ensure that the system works properly before and after its adaptation, the system requirements need to be validated. First, to ensure that adaptation actions decided by the system in response to a context change lead to a valid system state (variant), the adaptation requirements need to be checked for consistency [38]. To check the requirements consistency, we generate adaptation actions (as scripts) that can be applied to the system in response to runtime context changes. These scripts are then automatically checked for consistency to ensure that only valid actions are applied to the system at runtime.

Second, the functional requirements need to be validated with respect to a number of system properties to ensure that these properties are preserved while the system is in operation [23]. To validate the functional requirements, we automatically enumerate and generate the system variants from its scenarios. Then, we transform the variants to Petri nets [39] and the properties that need to be hold at runtime to computational tree logic formulas [40], so that the Romeo model checker [41] can be used for checking the variants against the properties.

System’s Design and Realization (Step 3): To design a context-aware adaptive system (see Figure 1-3), the functional requirements need to be mapped to a functionality model that capture the system structure and behaviour [42]. In addition, context information that is required by the system to continue its operation, or can trigger the system adaptation is used to define a context model [9]. Furthermore, an adaptive behaviour model needs to be designed to specify the system reactions to anticipated context changes [43]. To realize the software system, its design model is transformed to a deployable system [44]. The functional model is used for generating implementations of the system functionality, while the context model defines what the needed context providers are (i.e. the context acquisition). In addition, to adapt the system in response to context changes, the adaptive behaviour model is transformed to an implementation that manages the system while it is in operation (i.e. the system management).

To design a context-aware adaptive system, we introduce an organization-based meta-model where the system is modelled as an organization. In such view, the system is explicitly modelled from three aspects: functionality, context, and adaptive behaviour. As such, the aspects and their relationships can be clearly captured and easily changed at runtime. We adopt an organizational approach because it captures the relationships between the system elements explicitly. Thus, the system’s elements and their relationships can be clearly captured and easily manipulated. It also keeps the system’s structure alive at runtime, so that the system can be easily changed while in operation [45-46]. In addition, to maintain a causal connection between the system requirements and its design model, we support automatic synthesis of the system’s initial design model from
its scenarios. This model can be later completed by the software engineer to add elements that are related to the system’s solution space, but not possible to be synthesized from the scenarios. To realize the system, we transform the system’s design model to an executable model that is compatible with the ROAD framework [47]. When this model is deployed to the ROAD runtime environment, an instance of the system is created. This instance supports the application of different adaptation actions to the system where the deployed system artifacts are engineered with actions for manipulation (i.e. add, remove, or modify) at runtime (i.e. evolvable artifacts).

**System’s Runtime Evolution (Step 4):** To incorporate unanticipated changes, first, the system requirements specification needs to be changed. These changes are adding new functional (e.g. adding a new system feature) and adaptation (e.g. adding a system reaction to a context change) requirements, or modifying existing requirements [48]. To support this task, we enable runtime changes to the specified scenarios to add new requirements or to change existing ones.

Second, the functional and adaptation requirements of the system need to be checked to ensure their consistency and validity after introducing changes to them [49]. To validate the changed requirements, we use the technique described in Step 2 (see above). However, before using this technique, we identify those adaptation scripts and variants that need to be validated based on the requirements’ changes (i.e. only the changed parts are validated).

Third, the system’s design model needs to be modified to reflect the requirements’ changes [50]. For example, to add new context information to the system, the context model is modified by adding this information and a set of changes are performed to the system’s functional and adaptive behaviour models accordingly. Similarly, to consider new functionality, the system’s functional model is modified to include such functionality, while the context model is changed to add context information that is required by that functionality. Some changes are also injected into the adaptive behaviour model, so that the system can adapt while taking into account this new functionality. To ease the system design evolution, we introduce a set of evolution patterns. The patterns specify the changes to the system design in response to the changes in the system requirements. We use these patterns to automatically synthesize the changes to the system’s design model from the scenarios’ changes.

Fourth, the changes to the system’s design model need to be realized to the system while it is in operation [33]. To do so, we compute the differences between the running system model and its evolved model, and use the differences to generate adaptation actions that need to be applied to the system at runtime. The actions are then applied to the running system using the adaptation methods engineered into the system’s evolvable artifacts at the development time.
1.4 Research Contributions

The following are the original contributions of the thesis:

- A scenario-based technique to specifying the requirements of a context-aware adaptive system. It specifies the requirements as two sets of scenarios: functional and adaptation. In such scenarios, the context and its operational and management relationships with the system are explicitly represented. This technique also captures the system properties in a form similar to the scenarios.

- Techniques to check the consistency of a system’s adaptation requirements and ensure the validity of its functional requirements relative to a set of properties that need to hold while the system is in operation.

- An organization-based meta-model to enable the design of a context-aware adaptive system, and a technique to automatically synthesize a system’s design model from its requirements (specified as two sets of scenarios) and transform the design model to an executable model for the deployment.

- Techniques to enable the runtime evolution of a context-aware adaptive system:
  - A set of patterns that specifies changes to a system’s design model in response to unanticipated changes, and enables the automatic synthesis of the evolved design model of the system from its changed scenarios.
  - A technique to incrementally validate a system’s functional and adaptation scenarios, when they are changed to incorporate unanticipated changes.
  - A technique to realizing the changes of a system’s design model to a running instance of the system automatically.

1.5 Thesis Outline

The thesis is structured into three parts. The first part consists of the introduction (Chapter 1), a motivational scenario (Chapter 2), and an analysis of existing approaches (Chapter 3). Our approach is introduced in the second part. The approach supports the development of a software system that is able to cope with anticipated changes, and enables the system’s runtime evolution in response to unanticipated changes. It supports the following engineering tasks: a system’s requirements specification and validation (Chapter 4), synthesis of a system’s design model
from its requirements and transformation of this model to an executable model (Chapter 5), and evolution of the system to cope with unanticipated changes (Chapter 6). A graphical tool is also developed to support the approach (Chapter 7). Finally, the third part includes two case studies (Chapter 8), the approach evaluation (Chapter 9), and conclusions and future work (Chapter 10).

Chapter 2 provides a motivating scenario based on a software company that develops and evolves a travel guide system. This scenario is used throughout the remaining chapters of the thesis to explain the concepts of our approach. It is also used for identifying a number of general requirements that need to be considered in context-aware adaptive systems. In this chapter, we further identify a number of challenges that are related to the engineering tasks that need to be performed to develop a context-aware adaptive software system, and to enable the system’s runtime evolution to cope with unanticipated changes.

In Chapter 3, we analyse existing approaches relative to the engineering challenges identified in Chapter 2. For each approach, we describe the approach in general, the challenges that are tackled by the approach, and the limitations of the approach. The result of this analysis is a set of challenges that hinder easy and effective development and runtime evolution of context-aware adaptive software systems.

Our scenario-based technique to specifying a context-aware adaptive system’s requirements is presented in Chapter 4. The technique is an extension to the UML sequence diagram to specify the system requirements as functional and adaptation scenarios. The functional scenarios capture the system functionality, while the adaptations scenarios represent the system adaptation in response to anticipated context changes. The extension to the sequence diagram also enables the graphical representation of the system’s properties that need to hold at runtime in a form similar to the scenarios.

Chapter 4 also investigates the validation of the system requirements. First, to validate the adaptation requirements, adaptation actions (as scripts) that need to be applied to the system in response to runtime context changes are generated. Then, the scripts are validated to ensure their consistency. Second, we introduce a technique to automatically generate the system variants from its scenarios’ descriptions. We also identify the system properties that need to be checked against the variants. Finally, the variants and their properties are transformed to formal models, so that an existing model checker is used to check the variants against the properties formally.

Chapter 5 presents an organization-based meta-model to design a context-aware adaptive system. This meta-model supports the system modelling from three aspects explicitly: context, functionality, and adaptive behaviour. The system functionality captures the functions that need
to be provided by the system, while the context aspect represents the context information that is needed by the system to continue its operations or can trigger the system’s runtime adaptation. The system’s adaptive behaviour specifies how the system adapts itself in response to context changes. To ease the task of designing the system, we also introduce a technique to synthesize the design model of the system from its scenarios. Finally, we describe how to transform the system’s design model to an executable model to realize the system.

Chapter 6 presents a set of change patterns to enable the system evolution. These patterns specify the changes to a system’s requirements specification and its design model to cope with unanticipated changes. Using the patterns, the software engineer changes the system functional and adaptation scenarios to incorporate the unanticipated changes, and then we map the changes of the scenarios to the system design automatically. This chapter also describes how to validate the changes to the requirements specification, and realize these changes to the running system. To validate the changes, we identify the adaptation scripts and variants that are affected by the changed requirements. We also identify system properties that need to be checked against the changed variants. Then, the adaptation scripts are checked for consistency and the variants are checked against their properties using the techniques introduced in Chapter 4. To realize the changes, we compute the differences between the running system model and its evolved model. These differences are used for generating an adaptation script that contains actions that are then applied to the running system to realize the changes.

Chapter 7 describes the graphical tool we have developed to support the approach. This tool implements the introduced techniques and supports the software engineer in performing the engineering tasks that are needed for developing and evolving a context-aware adaptive system. In this chapter, we also provide a guide on how to perform the system development and runtime evolution using the developed tool.

Chapter 8 demonstrates the approach applicability by using it to develop and evolve a travel guide software system (described in Chapter 2), and an electronic exam system as systematic case studies. We also discuss the lessons learnt from doing the two case studies.

Chapter 9 evaluates the approach by carrying out four types of evaluations. First, we perform a feature-based analysis of the approach (i.e. a qualitative evaluation). To do so, we list features that an approach to developing and evolving context-aware adaptive systems needs to support, and then analyse how these features are supported by our approach systematically. Second, the effectiveness of the visual notations we use for specifying a software system’s requirements and for supporting its design is evaluated. Third, to assess the reduction of engineering effort that is
gained by using our approach to develop and evolve context-aware adaptive software systems, we estimate the efforts that are needed for the development and evolution of a software system with and without our approach. Then, the effort reduction is computed as the difference between the estimated efforts. Finally, a performance evaluation of the approach is performed by: (a) measuring the time required for performing a system’s runtime adaptation, and (b) analysing the computational complexity of the algorithms we introduced in the approach.

Chapter 10 concludes the thesis by summarising the original contributions of the thesis to the development and runtime evolution of context-aware adaptive software systems. Some topics for future investigations to further improve the approach are also outlined.
Chapter 2: Motivational Scenario

In this chapter, we present a motivating scenario based on a software company that develops and evolves a travel guide software system. The purpose of the scenario is threefold. First, we use the scenario to identify challenges that need to be considered in developing and evolving context-aware adaptive systems. Second, the scenario is used throughout the thesis chapters to explain the concepts of our approach by providing suitable examples. Third, the development and runtime evolution of the travel guide software system described in this chapter is later used as a case study to demonstrate the applicability of our approach.

The first section of the chapter discusses a number of general requirements that need to be taken into account by the company during the development and runtime evolution of the travel guide system. In Section 2.2, challenges that need to be tackled to support these requirements from the software engineer’s perspective are presented.

2.1 The Travel Guide Software System

In the following, we present a number of general requirements that need to be considered by a software company during the development and runtime evolution of a travel guide system.

2.1.1 Composing a Context-aware System

The travel guide provider develops a system that provides a number of services to the users such as route planning and finding attractions. The route planner plans a route from a location to a destination while taking the live traffic information into account. The attractions finder searches for nearby attractions and suggests a set of them to the user based on the weather forecast. This service is also integrated with the route planner, so that a route to visit a number of attractions can be suggested.

As such, the travel guide system needs to be composed of a set of functional services (e.g. route planner and attractions finder) that interact with each other to meet the users’ needs, while considering some quality requirements (e.g. fast route planner). In addition, these services need to take the context information (e.g. the weather and the traffic information) into account with a certain quality (if needed) to give better suggestions to the users. For example, the route planner needs the live traffic information to provide accurate estimations for the possible routes’ travel times, and to display the routes that are less congested.
2.1.2 Runtime Adaptability to Cope with Anticipated Changes

The travel guide system needs to be developed with adaptability in mind, so that the system can be customized to suite different anticipated context situations. First, the system provider wants the users to pay for its services, and then different levels of memberships are available for using the system services. These memberships allow the users to use the route planner service with a cost of 5$/month while they need to pay 2$/month for the attractions finder. As such, if the user chooses the route planner service to be only included in his subscription, the travel guide system will only have this service as shown in Figure 2-1.A.

While the travel guide system is in operation, the user may want to include the attractions finder service. To include such service, several changes are applied into the running system. The system is first adapted by adding the attractions finder service (i.e. adding a functional service). Then, to find suitable attractions for the user, there is also a need to get the weather information and use it in searching for and suggesting the attractions (i.e. changing the system by including context providers and their relationships with the system functionality). Figure 2-1.B shows the travel guide system after adding the attractions finder service and the weather information, so that the system can find attractions that match the weather forecast.

Figure 2-1: Adapting the travel guide system to include the attractions finder service
Second, the system needs to be adapted at runtime in response to changes in the availability of the context information, so that it can work properly without the unavailable information. For example, the travel guide has two route planners, where the route planner used when the traffic information is available is different from the one used when this information is not available.

2.1.3 Runtime Evolution in Response to Unanticipated Changes

To increase users’ satisfaction, the travel guide system should be available 24/7 (24 hours a day, 7 days a week). While the system is in operation, a new source of context information becomes available such as the speed limit and the system provider wants to use this information to alert the driver when his vehicle speed exceeds the speed limit. In addition, the provider wants to add a restaurants locator service to the system to attract more users and compete with providers of the same system in the market. This service notifies the user about nearby restaurants that match his food preferences. As such, the travel guide system needs to evolve at runtime in response to unanticipated changes such as including new context information (e.g. the speed limit), and adding new functionality (e.g. the restaurants locator).

2.2 Challenges for Engineering Context-aware Adaptive Systems

To develop and evolve context-aware adaptive software systems (e.g. the travel guide system) while considering the general requirements discussed in Section 2.1, a number of challenges from the software engineer’s perspective need to be considered. These challenges are related to the engineering tasks that need to be performed during the development and runtime evolution of a context-aware adaptive system. We classify the challenges into four groups according to the different engineering tasks. The first group includes challenges that need to be considered in specifying the system requirements, while the second group contains a set of challenges that are related to the validation of the system requirements. The challenges to be considered during the system’s design and realization are described in the third group. The fourth group consists of challenges that are related to the system’s runtime evolution.

2.2.1 Specifying the System Requirements

The success of a software system is measured by the degree it meets its requirements [51]. As such, specifying the system requirements is an important task. To specify the requirements of a context-aware adaptive system, a number of aspects need to be considered as follows.

(1) Functional Requirements: The functional requirements specify functions that need to be provided by the system (e.g. route planning). These functions also take the context information
into account to give better suggestions to the users [35]. For example, the route planner needs the live traffic information to provide accurate estimations for the routes’ travel times.

(2) **Adaptation Requirements**: While the software system is in operation, it needs to adapt itself in response to anticipated context changes to keep achieving the user needs [36]. We call such system reactions to context changes as (anticipated) adaptation requirements. For example, while the travel guide system is in operation, the user may want the attractions finder service that is not provided to him initially. To allow the user interactions with this service, the system is adapted by incorporating the attractions finder service. Another adaptation requirement is the system reactions to changes in the status of the traffic information availability.

(3) **Quality Requirements**: The user may want the system functionality to be performed with a certain quality. For example, the route planning function should be performed in less than five seconds. Therefore, the system’s *functional qualities* (i.e. non-functional requirements) need to be specified [52]. Similarly, the system needs to take the context information into account while considering its quality. For example, the route planner needs the traffic information up to the last minute. Therefore, the *context qualities* need to be also captured [53].

(4) **Environment Uncertainty**: The system is usually deployed into an environment that is not totally anticipated at the development time. This environment uncertainty affects the satisfaction of the system’s functional requirements at runtime [54]. As such, the environment uncertainty needs to be taken into account during the functional requirements specification [55].

### 2.2.2 Validating the System Requirements

The adaptation requirements specify changes to the system functionality in response to context changes. To ensure that the system works properly before and after its runtime adaptation, some aspects of the requirements need to be validated. The following are the challenges that need to be considered in validating the system requirements.

(1) **Specifying the System Properties**: While the system is in operation, a number of system properties need to hold to ensure that the software system works properly [56]. These properties are either *local* properties that need to be preserved in a specific context situation (e.g. the user cannot use the attractions finder service when it is not included in his subscription), or *global* properties which need to hold in all context situations (e.g. the user should login to the system before using its services). Thus, the system’s local and global properties need to be specified.

(2) **Validating the Adaptation Requirements**: In response to a context change, a set of actions are applied to the functional requirements. These actions need to be consistent with each other,
so that their application leads to a well formed set of functional requirements [38]. The actions are **consistent** if they are free from errors such as actions’ **redundancy** (i.e. an action is triggered twice in a context situation), **conflict** (i.e. conflicting actions are fired in response to context changes), and **incompleteness** (i.e. some adaptation actions need to co-exist with each other, and then the application of an action to the requirements specification without applying the other action that must co-exist with it leads to an invalid set of requirements). Therefore, the system’s adaptation requirements need to be checked for such types of errors to ensure that the adaptation actions specified in such requirements are consistent.

(3) **Validating the Functional Variants**: At runtime, the system switches from one variant (a set of functional requirements) to another in response to a context change by applying a set of adaptation actions into the system functionality. If the adaptation requirements are consistent, it does not mean that the functional requirements are also valid where applying a set of adaptation actions to the system to cope with a context change may violate a number of system properties. In addition, due to the high variability of the system, it has a large number of variants that suit different context situations. Thus, the large number of the system’s functional variants need to be identified (enumerated) and validated with respect to the system properties [23].

2.2.3 Designing and Realizing the System

After specifying and validating the system requirements, a model for the system needs to be designed. This design model has elements that provide the specified requirements. Executable code needs to be also created from the system’s design model. This code provides the system functionality that takes the context information into account to give better suggestions to the users. It also enables runtime changes to the system in response to anticipated and unanticipated changes. To **design** and **realize** a context-aware adaptive system, a number of challenges need to be considered. In the following, we describe these challenges in detail.

2.2.3.1 Designing Context-aware Adaptive Systems

In designing a context-aware adaptive system, there is a need to take a number of perspectives (aspects) into account.

(1) **System Functionality**: The system functionality is functions that need to be provided by the system. In order to design the system functionality, two perspectives should be considered: **structure** and **behaviour** [42]. The system structure defines the system elements and their relationships [57]. The system behaviour specifies sequences of interactions between the system elements to provide its functionality [58]. Also, the system elements and their relationships need
to be explicitly represented, so that the elements and their relationships can be clearly captured and easily manipulated (adapted) at runtime.

(2) **Context Model**: Context is the information about entities that affect the system operations [9]. Two types of contexts need to be considered: *functional* and *management*. The functional context is the information that is needed by the system to operate effectively (e.g., suggesting attractions that match the weather forecast) [11]. The management context is the information that triggers the system’s runtime adaptation (e.g., changes in the status of the traffic information availability trigger the system adaptation, so that the system can work properly with and without the traffic information) [59]. Consequently, a model for the context needs to be specified.

(3) **Adaptive Behaviour**: In response to context changes, the system needs to be adapted. As such, the system reactions to context changes need to be captured in the system’s design model (i.e., an adaptive behaviour model) [43]. The adaptive behaviour model specifies how the system adapts from a configuration/behaviour to another to keep achieving the user needs (e.g., adding the attractions finder service when it is needed by the user).

(4) **System Constraints**: To ensure that a context-aware adaptive system is working properly, a set of constraints need to be preserved at runtime [60]. These constraints need to be specified at the development time on the system’s structure and behaviour models. Similar to the system properties, these constraints can be local constraints (i.e., they need to be preserved in a specific system variant), or global constraints which do not depend on a system variant(s).

(5) **Validating the System Design**: To ensure that a system’s structure is well formed, it needs to be checked against a set of structure constraints [28]. Similarly, the system behaviour needs to be validated relative to temporal constraints that need to hold during the system execution [61]. In addition, the system adapts at runtime in response to context changes, and then it has multiple variants (i.e., structure and behaviour variants). As such, these different variants need to be enumerated and checked against their constraints.

(6) **Managing the Design Complexity**: A context-aware adaptive system has multiple aspects: functionality, context, and adaptive behaviour. To clearly capture the system’s design model, these aspects and their relationships need to be explicitly captured [62]. In addition, modelling a large scale system is complex and error prone task, and then the system needs to be designed at different levels of abstraction [63]. As such, the system at a higher level can be simple, while the details are specified at lower levels.

(7) **Design Synthesis**: To maintain a causal connection between a context-aware adaptive software system’s requirements and the different aspects of the system design and to reduce the
effort required for designing the system, the system’s design model needs to be derived from its requirements automatically [37]. This design model is then completed by the software engineer to add elements that are related to the system’s solution space but not possible to be synthesized from the system requirements.

2.2.3.2 Realizing Context-aware Adaptive Systems

To realize a context-aware adaptive system, the system design model needs to be transformed to an implementation. This implementation should support the application of different adaptation actions to the running system in response to anticipated and unanticipated changes [64]. In the following, we describe the challenges that need to be considered in realizing the system.

1. **Automatic Realization**: A context-aware adaptive system has a high variability because it operates in a dynamic environment. In addition, the system has a number of aspects that are highly related to each other (as discussed above). As such, it is a difficult task for the developers to implement the system while considering its runtime adaptability. Model-driven development is the notion of constructing a model for the system which can be realized automatically [44]. Consequently, a model-driven approach is promising for realizing the system, where it eases the developers’ task by generating part of the system implementation from its design model.

2. **Runtime Flexibility**: At runtime, the system needs to be adapted either to keep achieving the user needs in the face of **anticipated** changes, or in response to **unanticipated** changes. The degree of the system flexibility is measured by the number of system aspects (elements) that can be changed at runtime [65]. The greatest degree of flexibility is that all the system aspects (i.e. functionality, context, and adaptive behaviour) are changeable while the system is in operation to cope with the anticipated and unanticipated changes.

2.2.4 The System’s Runtime Evolution

While a system is in operation, it needs to **evolve** in response to unanticipated changes in its environment or requirements [66]. Below, we describe the challenges related to the engineering tasks that enable the system’s runtime evolution.

1. **Specifying the Changes**: The system provider may want to add new functionality (e.g. the restaurants locator) or to take new context information into account (e.g. the speed limit) [48]. In response to such unanticipated changes, the requirements specification of the system and its design model need to be changed. First, the functional requirements may change by adding a functional requirement or by modifying existing ones, while the changes to the adaptation
requirements are to include a reaction to a new context change or to change the way that the system currently reacts to an anticipated context change. Second, the system’s design model is modified to reflect the changes in the system requirements [50]. The design model evolves by introducing new elements to the system’s three aspects (i.e. functionality, context, and adaptive behaviour) or by changing its existing elements. In a large scale system with high variability, the above task is difficult where the changes and their effects on the requirements specification and the design model of the system need to be identified and specified in an ad-hoc manner. To ease this task, a technique is needed to assist the engineer in evolving the system’s requirements specification and design model in response to the unanticipated changes.

(2) Validating the Changes: As the system’s requirements specification evolves, a validation of the changed requirements is required to ensure the consistency of the adaptation requirements and the functional requirements validity [49]. In addition, to ensure that the system constraints are still preserved in the changed design model of the system, the changed design model needs to be validated [67]. Validating the whole requirements specification and the design model of the system is inefficient and time consuming task. Therefore, a technique is needed to identify and only validate the changes to the system’s requirements and design model.

(3) Realizing the Design Changes: After changing the system’s design model in response to the unanticipated changes, the design changes need to be applied to the system at runtime where the system should be available 24/7 [68]. Applying the changes to the running system manually is a tedious and an error prone task. Thus, a technique is needed to realize the design changes to the running system automatically [33].

2.3 Summary

In this chapter, we have presented a motivational scenario based on a software company that develops and evolves a travel guide software system. Based on this scenario, we have identified a set of general requirements and engineering challenges that need to be considered during the development and runtime evolution of a context-aware adaptive software system.

First, the general requirements that need to be taken into account at the system development and evolution are: (1) the system needs to be composed of functional services that provide the system functionality while taking the context information into account to give better suggestions to the users; (2) while the system is in operation, it needs to adapt itself in response to changes that were anticipated at the development time; (3) the system needs to evolve at runtime to cope with unanticipated changes in its environment and requirements.
Second, to develop and evolve the system while considering the general requirements, a set of engineering challenges need to be tackled during the system’s development time and runtime. At the development time, there is a need for techniques to specifying and validating the system requirements, designing the system model based on its requirements, and realizing the system. At runtime, to enable the system runtime evolution, a set of techniques are needed to specify the changes to the system requirements specification and its design model, validate the changes, and realize the design changes to the running system automatically.
Chapter 3: Literature Review

Recently, a number of approaches have been introduced to assist the software engineer in the development and runtime evolution of context-aware adaptive software systems. In this chapter, we review and analyse these approaches with respect to the challenges identified in Chapter 2.

A classification of existing approaches for developing and evolving context-aware adaptive systems is discussed in Section 3.1. The analysis of the existing approaches with respect to the challenges identified in Chapter 2 is presented in Sections 3.2-3.5. These sections also present a summary of the approaches, and identify the challenges that still hinder the development and runtime evolution of context-aware adaptive systems. In Section 3.6 we present, in general, how our approach tackles these challenges.

3.1 Classification of Approaches for Developing and Evolving Context-aware Systems

In recent years, a number of approaches have been introduced to assist the software engineer in developing and evolving context-aware adaptive software systems. Due to the large number of approaches that have been introduced in the past, this chapter cannot cover all of them. Thus, we only analyse a sufficient number of approaches based on their contribution, originality, and practically. Figure 3-1 shows our categorization of such approaches. We classify the approaches into four groups relative to the engineering tasks that are supported by them.

The first group contains a set of approaches that support the specification of a context-aware adaptive software system’s requirements. We group these approaches based on the requirements specification language that they have extended to capture the system’s functional and adaptation requirements. The existing approaches have extended i* language [69] or its extension Tropos notation [70], a goal modelling language (e.g. [71]), or the problem frames approach [72] (see Figure 3-1).

The second group has a set of approaches that assist the software engineer in validating the requirements of a context-aware adaptive software system. These approaches are classified into two categories (i.e. specifying the system properties and validating its requirements) as shown in Figure 3-1. First, to specify the system properties, formal languages (e.g. linear temporal logic [73]) or high level representations of these formal languages have been used (e.g. property specification charts [74]). Second, we classify approaches that enable the system requirements’ validation based on which requirements aspect is validated. These approaches either support the validation of the system’s functional requirements (i.e. the system variants) with respect to a set
of system properties (e.g. [23]), or enable the validation of its adaptation requirements to ensure their consistency (e.g. [25]).

![Diagram](image)

**Figure 3-1:** A classification of existing approaches based on the engineering tasks

The third group contains approaches that enable the design and realization of context-aware adaptive systems. The existing approaches either concerned with designing the system structure and enabling its runtime adaptation to suit different context situations (e.g. [16, 34]), or focus on specifying the system behaviour and supporting its adaptation in response to runtime context changes (e.g. [27, 75]). These approaches also enable the system realization either by providing a middleware/platform that can be extended to realize the system (e.g. [76-77]), or by using a model-driven approach that automatically transforms the system’s design model to executable model/code (e.g. [28, 78]) as shown in Figure 3-1.

The fourth group includes approaches that enable a system’s runtime evolution to cope with unanticipated changes in its environment or requirements. These approaches are concerned with specifying the required changes to the system specification and its design, or with realizing the changes. First, the changes to the system are specified either in an ad-hoc manner (e.g. [29]), or using a set of patterns that guide the software engineer in doing this task (e.g. [50, 79]). Second, approaches for realizing the system changes are grouped into two categories: code-based and script-based. In the code-based approaches, the changes are incorporated into the system source code, and then they are synchronized with the running system (e.g. [32, 68]). The script-based approaches enable the engineer to write a set of high level actions (e.g. add component) that can be directly executed into the running system (e.g. [8, 62]).
3.2 Specifying the System Requirements

To specify a context-aware adaptive system’s requirements, a number of approaches have been introduced. We classify them into three categories based on which requirements specification technique (e.g. a goal modeling language, i* language or its extended Tropos notation, or the problem frames approach) has been extended. In the following, we discuss these approaches and identify their limitations.

3.2.1 Approaches based on General Goal Models

RELAX is an approach to capture the requirements of an adaptive system through relaxing the system requirements to take into account context (environment) changes [54]. They first capture the system requirements (goals) as SHALL statements in a textual representation. Then, they classify these requirements as relax-able (i.e. requirements that are changeable at runtime to take context changes into account) or invariant (i.e. requirements that cannot be violated at runtime) requirements. Finally, they introduce RELAX operators (e.g. as early as possible) to the relax-able requirements, so that they become flexible to cope with runtime context changes. The relax-able requirements also specify context information that causes the relaxation of such requirements through defining four factors: ENV (environment) that defines what information need to be considered, MON (monitor) that specifies how the information is monitored, REL (relationship) that specifies the relationship between ENV and MON, and DEP (dependency) that specifies the impact of relaxing a requirement into other requirements. An example requirement from a smart office system is “The synchronization between a user devices SHALL be initiated when he enters his office and at 30 minute intervals after that” [54]. Due to problems in communication links and characteristics of the user devices, this requirement may not be satisfied and then it needs to be relaxed to take into account environment changes. The relax-able version of this requirement is:

“The synchronization between the user devices SHALL be initiated AS EARLY AS POSSIBLE AFTER the user enters his office and AS CLOSE AS POSSIBLE TO 30 minute intervals after that.

ENV: User location; Synchronization interval.
MON: Location sensor; Network sensors.
REL: Location sensor provides the user location;
Network sensors provide synchronization interval”.

The RELAX approach focuses on relaxing a system’s requirements in response to context changes, without considering the use of the context information to give better suggestions to the users. In addition, specifying requirements of a large system using this approach is tedious and error prone task due to lack of tool support and the textual representation of the requirements.
Lapouchian et al. proposed an approach to capture alternatives that an adaptive system can have in response to changes in its context and/or the user needs [80]. To represent the system alternatives, they adopt a goal modelling language [71]. In such language, the system goals are related to each other by AND/OR relationships. The AND relationships specify that all goals need to be achieved, while the OR relationships mean that one of the goals need to be achieved. They use the OR relationships to specify the system alternatives. An example goal model for a meeting scheduling system is shown in Figure 3-2. It has the different system alternatives where the time tables can be collected by a person or by the system (VP1), the meeting schedule can be specified manually or automatically by the system (VP2), and the time tables are collected by the system through agents or from the users directly (VP3).

This approach focus is specifying the variability (alternatives) of a system’s requirements by a goal model. However, it does not focus on how to switch between the system’s alternatives to cope with context changes, and how the context information is used by the system functionality.

Baresi et al. proposed an approach to capture the goals of an adaptive system [20, 81]. They capture the system’s functional requirements by a goal modelling language. In this approach, the system goals are classified as crisp-goals where their satisfaction can be expressed as yes or no (i.e. 0 or 1), or as fuzzy-goals which have different degree of satisfaction between 0 and 1. For example, in a washing machine system, the goal “add powder” is either achieved or not (i.e. a crisp-goal) while the goal “maintain low energy” can be satisfied with different degrees (i.e. a fuzzy-goal).
fuzzy-goal). The fuzzy goals are specified by the operators introduced in the RELAX language [54]. The adaptation requirements can be defined based on the goals satisfaction, where a set of actions are specified when a goal is not satisfied. For example, in high energy consumption (i.e. the “maintain low energy” goal is not satisfied), the system uses an economic program for washing the clothes.

This approach claims that it has the ability to specify a system’s adaptation requirements. However, the language that is used to specify these requirements is not presented. In addition, in specifying the functional requirements, it does not consider the functional use of the context.

YunSong et al. introduced a goal-based approach to capture the requirements of an adaptive system [82]. In this approach, the requirements are represented as 4-tuple: <G, E, A, R>. “G” is a set of system goals, “E” is environment entities that trigger the system adaptation, “A” is activities (tasks) to be performed to achieve the system goals, and “R” is adaptation rules that specify which activities are used to achieve the system goals in specific context situations. For example, a system may have four goals (i.e. “g0, g1, g2, and g3”), an environment entity “e1” that affects its operations, a set of activities (i.e. “a1, a2, a3, and a4”) that are used to achieve the system goals, and two adaptation rules. The first adaptation rule assigns the activity “a2” to the goal “g2” when the entity “e1” has the value “C1”, while the second rule assigns the task “a3” to the goal “g2” when the environment entity “e1” holds the value “C2”.

In this approach, a system’s functional and adaptation requirements are specified into one model. Thus, this model becomes complex when a large number of system reactions to context changes need to be specified. In addition, the system adaptation in this approach is limited to the selection between different tasks (activities) to achieve a system goal.

An approach to represent the control loop of adaptive systems as a pattern in a goal-oriented model is introduced by Nakagawa et al. [83]. In this approach, the functional and adaptation requirements are expressed using a KAOS (Keep All Objectives Satisfied) goal model [24]. The model consists of a set of goals that are related to each other by AND/OR relationships. The goals are provided by (assigned to) agents that can be persons, devices, or softwares. The agents are also able to monitor a number of entities by acquiring information about them. In the same manner, the goals can be aware of some environment entities.

To represent the control loop in the goal model, the four tasks of the loop (i.e. collecting, analysing, planning, and acting) are assigned to a set goals in the requirements model. First, a goal is defined that is responsible for collecting the context information that triggers the system adaptation. Second, a goal is specified to capture the analysing and deciding operations. Third,
the result of the deciding operation is acted to the system by a set of goals that are responsible for applying the decided actions. These goals represent a control loop pattern in the goal model. Following this pattern, the system reactions to context changes are specified as a set of control loops in the goal model.

This approach represents the control loop explicitly in the goal model. But, representing the functional and adaptation requirements into one model makes the goal model complex in the case of having a large number of reactions to context changes. It also becomes hard to manage the goal model graphically.

**Adapt Cases** is an approach to modelling a system’s adaptivity at the logical design phase to fill the gap between the system’s analysis and design [84]. In this approach, the adaptation requirements are specified using a goal-based approach (e.g. RELAX [54]). Then, these goals are refined manually into one or more concrete adapt cases. Each adapt case is specified as If-Then-Else rules textually. These rules define the system reaction(s) to context changes. Example adapt cases are shown in Figure 3-3. They are used to adapt the znn.com system (described in [85]) based on the response time of the users’ requests and the server load capacity. First, when the system’s response time is high, the system increases the number of servers or switches to the textual mode if the maximum number of servers is used (see Figure 3-3.A). Second, in the case of the number of running servers is more than the needed to handle the current users’ requests, the system either switches to the multi-media mode, or deceases the number of servers when it is already in the multi-media mode as shown in Figure 3-3.B.

The language that introduced to specify the adaptation requirements is a textual language, and it does not have a tool support. Thus, the task of specifying the adaptation requirements for a large scale adaptive software system is tedious (complex) and error prone.
Souza et al. proposed an approach to capture requirements that trigger the system adaptation which they call awareness requirements (or AwReqs, for short) [19]. They capture the system requirements using a goal-oriented approach. The awareness (monitoring) requirements are then specified as success or failure of the system requirements. There are five types of awareness requirements. First, the simplest type is “never fail requirement” which means the goal should be always satisfied (e.g. the goal “input emergency information” shown in Figure 3-4). Second, the “aggregate requirement” is a goal that needs to be satisfied with a percentage. For example, the goal “search database” needs to have a success rate of 95% in seven days as shown in Figure 3-4. Third, the success of a goal may need to increase (or decrease) from a month to another, and then it is called “trend requirements”. Forth, a goal may need to be achieved in a specific period of time such as a task needs to be performed within 5 minutes (i.e. “delta requirement”). Finally, an awareness requirement can be about another awareness requirement as shown in at the right of Figure 3-4 (i.e. “meta requirement”). This approach also supports the formalization of the AwReqs [86] to enable the runtime monitoring of such requirements using the EEAT (Event Engineering and Analysis Toolkit) [87].

![Figure 3-4: Graphical representation for the awareness requirements (Cf. [19])]()

To capture a system’s adaptation requirements, there a need to represent the requirements of the monitoring, deciding, and acting operations. But, this approach only captures the monitoring requirements without paying attention to the deciding and acting operations.

### 3.2.2 i*/Tropos-based Approaches

**LoREM** is an approach to specify the requirements of an adaptive system [88]. It captures the system functionality as multiple non-adaptive systems, and each system is executing at a given point of time. In response to a context change, the system switches from a state (i.e. the source system) to another (i.e. the target system). The requirements of the non-adaptive systems are modelled by the $i^*$ notation [69]. An example system specified using this approach is the flood...
warning system. This system has two states: S1 and S2. The state “S1” is used when the river is quiescent. In that case, the system’s energy efficiency has a higher priority than the predication accuracy of the flood. As such, the river flow rate is calculated using a single host node. When the flow rate increases, the predication accuracy is more important than the energy efficiency, and then multiple nodes are used for calculating the river flow rate (i.e. the state “S2”). To capture the system adaptation in response to context changes, they have used $i^*$ notation to specify three aspects: what context information to be monitored, what changes in the monitored information trigger the system adaptation, and how the adaptation is executed. An example switching is between the states “S1” and “S2” of the flood warning system. In this switching model, the monitoring mechanism senses the flow rate. The decision maker checks whether the flow rate is over the normal level. When it is over the normal level, the adaptation mechanism replaces the single node processing with the multiple nodes processing.

In this approach, the functional requirements are specified as N states (non-adaptive systems) to suit N context situations. Thus, in a large scale system, a large number of system states and the switching between them need to be specified by the engineer which is a complex task.

To capture the requirements of self-adaptive systems, Morandini et al. [89] extended the Tropos framework [70]. First, they identified three types of goals: maintain, achieve, and perform. For example, in a dust cleaning system the goal “deal with dust” is a maintain goal, the goal “empty full dust box” is an achieve goal, and the “find dust” goal needs to be performed. Second, to represent environment entities that cause system adaptations, the UML class diagram is used where the entities are represented as a set of classes that are related to the system goals using a set of conditions. For example, the goal “empty full dust box” is enabled when the dust box is full, and then there is a “creation condition” that relates this goal with the cleaner agent’s dust box. This goal is also achieved only when the dust box is empty which is expressed by an “achievement condition”. Third, the Tropos notation is extended to model the expected system faults (errors), and the corrective actions in response to such faults.

Following this approach, the adaptation requirements of a context-aware adaptive system are intertwined with its functional requirements specification, and then the adaptation requirements cannot be clearly captured. In addition, the functional use of the context information is not considered in this approach.

Qureshi et al. introduced an approach to capture adaptation requirements of a system [21, 90]. In their approach, a model following the Tropos notation is used to capture the system’s functional requirements and an ontology model is used to express the context information that
triggers the system adaptation. These two models are linked by a set of relationships between
the system and the context changes. An example travel guide system specified by this approach
is shown in Figure 3-5. The right part of the Figure specifies the goals that need to be achieved
by the travel guide system, while the left part expresses the context information that is related to
the system. The link between these two parts is performed through numbers. For example, the
“detect device” goal is linked with the computer and phone devices by the numbers “2” and “3”
where this goal is to detect the devices information. To specify the system reactions to context
change, they specify adaptation points in the system’s behaviour specification. These points
select between alternative tasks to be executed based on the context situation. An example
behaviour specification is to send a booking confirmation to the user. It starts with getting the
type of the user device. Then, the booked itinerary is retrieved and sent to the user as a SMS, an
e-mail, or a fax based on the user’s device type.

In this approach, the system adaptation in response to context changes is hardwired with its
functional behaviour specification, and then specifying the system behaviour becomes a tedious
and an error-prone task. In addition, the adaptation actions are limit to the selection between
alternative tasks (i.e. the actions to add and remove a task/goal are not supported).

Ali et al. introduced an approach to capture relationships between a system’s goals and its
environment [91-92]. In their approach, the system’s functional requirements are captured using
the Tropos notation as a set of goals. Then, the requirements model is parameterized using a set
of context situations. These situations are specified in a hierarchical way, where each context situation is a composition of simple conditions on context attributes. The situations specify: (1) which system goal need to be achieved in an OR-decomposition (e.g. a goal “G1” need to be achieved in the context situation “C1” while the goal “G2” is achieved in the situation “C2”); (2) a goal activation (e.g. a goal “G3” is only activated in the context situation “C3”); (3) which task is used to achieve a goal (e.g. a task “T1” performs a goal “G4” when the context situation “C4” is true).

Similar to the above approaches, embedding the adaptation requirements with the functional requirements increases the requirements model complexity. In addition, the context situations are used for the selection between different elements of the requirements model. As such, they cannot be used for adding (removing) elements to (from) the requirements model.

### 3.2.3 Problem Frames-based Approaches

An approach is introduced by Salifu et al. to capture a system’s monitoring and switching requirements [93]. The monitoring requirements are the context variables that cause the system adaptations, while the switching requirements specify the system switching between its states to keep achieving its goals in the face of context changes. To capture the context information and its effect into the system, they adopt the problem frames approach [72]. First, the system states are specified. For example a picture transmission system specified using this approach has two states: S1 and S2. The system is in the state “S1” when the transmission is secure, while it is in the state “S2” in the case of the transmission is insecure. The difference between the two states is an element for encrypting and decrypting the transmitted content. This element is added to the system when the transmission is insecure and it is removed otherwise.

Second, context information that causes the system’s runtime adaptation (i.e. the monitoring requirements) is also specified by the problem frames. An example monitoring requirements is “a request is secure or not”. A request is considered secure based on the location it comes from, where the request is secure if it comes from a location that does not exist in the list of insecure locations.

Third, to specify the switching between the system states in response to context changes, a context switcher is specified using the problem frames. The switcher model contains the system states (i.e. S1 and S2) and the context information that triggers its adaptation. It also specifies when to switch between the system’s states in response to the context changes. For example, it switches from “S1” to “S2” when the system is insecure and from “S2” to “S1” otherwise.
In this approach, the system is specified as multiple states that suit different contexts. In a large scale system, a large number of system states need to be specified and a large number of transitions between these states need to be also defined. Therefore, specifying the requirements of an adaptive system by this approach is difficult and an error prone task.

3.2.4 Summary

As discussed above, a number of approaches have been introduced in recent years to specify the requirements of context-aware adaptive systems (summarized in Table 3-1). However, they still have several limitations.

1- Specifying the Functional Requirements: A context-aware adaptive software system needs to take the context information into account to give better responses to the users. But, existing approaches (e.g. [54, 84, 91, 93]) do not explicitly represent the context information and its functional use. In addition, the existing approaches specify the functional requirements as a set of goals. However, the stakeholders often find it difficult to articulate their needs as a number of goals [94]. As such, it is difficult to use these approaches to specify the functional requirements of a context-aware adaptive system while taking the context information and its functional use into account.

2- Specifying the Adaptation Requirements: Most of the existing approaches support the specification of a system’s adaptation requirements. But, they still have a number of limitations. First, in some approaches (e.g. [82, 89-91]), the system adaptation requirements are intertwined with its functional requirements specification, and then specifying the system requirements becomes tedious and error prone task. Second, a number of approaches (e.g. [88, 93]) specify the system as a set of non-adaptive states and context changes trigger the transitions between these states. Thus, for a large scale system with high variability, a large number of non-adaptive states and the transitions between them need to be specified manually which is a difficult task.
Third, a number of approaches (e.g. [80-81]) support a limited number of changes that can be performed on the system (i.e. only the selection between alternative goals/tasks is allowed), resulting in limited system flexibility.

3- Specifying the Quality Requirements: Two types of qualities need be captured: functional and context. However, existing approaches (e.g. [19, 83, 90]) focus on specifying the functional qualities of the system (e.g. response time, availability, etc.) with less attention to specifying the context qualities (e.g. freshness, accuracy, etc.).

4- Environment Uncertainty: The system is deployed into an environment that is not totally anticipated at the development time. To take the environment uncertainty into account, some approaches (e.g. [54, 81]) relax the system’s functional requirements (i.e. making them flexible) to consider this uncertainty. However, they do not provide a technique to specify the adaptation actions that need to be taken by the system in response to such uncertainty at runtime.

5- Tool Support: A software tool that supports a system’s requirements specification is needed to increase the adoption of an approach in practice. But, some of the existing approaches do not have tool support (see Table 3-1), which hinders their adoption for specifying a context-aware adaptive system’s requirements.

3.3 Validating the System Requirements
The functional and adaptation requirements of a context-aware adaptive software system need to be validated. First, the system’s functional variants need to be checked against system properties (that need to hold at runtime) to ensure the conformance of the functional requirements to these properties. Second, the adaptation requirements specify runtime changes to the system to cope with context changes. As such, they need to be checked for consistency to ensure that only valid adaptation actions are decided and applied to system at runtime. In the following, we analyse approaches that have been introduced for specifying a system’s properties, validating a system’s functional requirements with respect to the system properties, and checking the consistency of a system’s adaptation requirements.

3.3.1 Specifying the System Properties
To specify a system’s properties, a number of formal notations have been introduced such as linear temporal logic (LTL) [73], computational tree logic (CTL) [40], probabilistic CTL [95], finite state process algebra (FSP) [96], communicating sequential processes logic (CSP) [97], etc. However, system properties that can be easily expressed in natural language are hard to be
captured by such formalisms and writing such formalisms is tedious and error prone task [98].
Thus, a number of approaches have been introduced to tackle such problems by providing high
level representations of these formalisms. In the following, we describe some of them.

Dwyer et al. introduced the **property specification patterns** [56]. These patterns provide a
high-level easy-to-use notation for defining commonly occurring system properties that would
otherwise require formalisms such as computational tree logic (CTL). The patterns are classified
into two groups: occurrence and order. The occurrence patterns require some events to occur or
to not occur. The order patterns define the order for a set of events. Example occurrence patterns
are “Absence” that means an event does not occur during a system’s execution, and “Existence”
which means an event must occur within a scope. Example order patterns are “Precedence” that
means an event should always be preceded by another event during the system execution and
“Response” that means an event must be followed by another event. In addition, each pattern
has a scope to define when the pattern must hold. For example, the scope “global” specifies that
a pattern must hold during the entire system execution, while the scope “after” means a pattern
holds after the first occurrence of an event. Furthermore, each pattern specifies the mapping of a
system property specified by this pattern to a formal specification language such as LTL, CTL,
and quantified regular expressions (QREs) [99]. This approach eases the task of specifying a
system’s properties. However, it still require that the system to be specified using a state-based
(e.g. Petri nets [39]) or an event-based (e.g. labeled transition systems (LTS) [100]) model,
which hinders the adoption of this approach in practice.

VTS (Visual Timed event Scenarios) is a graphical notation that is used for capturing the
properties of real time systems to avoid the use of formal languages such as TCTL [101-102]. A
summary of the notation is shown in Figure 3-6.A. Each property specified by such notation has
a beginning, an end, a set of events, and relationships between the events. An arrow specifies
that an event “p” should occur before an event “q” (i.e. p precedes q). A dotted line defines the
events that are forbidden between two events (i.e. events restrictions). A temporal distance (a
time bound) may be also specified under a line that connects two events. A small full or open
circle connected to an event captures the last or the first event in a set of events respectively. An
open arrow near an event specifies that the event is forbidden between the two events connected
by the arrow (see next, previous, and consecutive relationships in Figure 3-6.A).

Smith et al. introduced the **TimeLine notation** [103]. This notation enables the graphical
representation of a system property as a timeline. The timeline is presented as a horizontal line
and the time progresses from left to right as shown in Figure 3-6.B. The timeline has vertical
lines (called marks) that define the events’ occurrences. These events are either regular events
which are optional events and donated by the letter “e”, or required events (donated by the letter “r”) that should occur if all previous events on the timeline have been occurred. The timeline also has a set of constraints that are specified as horizontal lines beneath the vertical bars of the timeline. The constraints specify forbidden events between the events defined on the timeline. An example constraint is “! Disconnect Incoming Call” shown in Figure 3-6.B. This constraint specifies that the tone event should hold as long as the incoming call is not disconnected.

To ease the task of specifying a system’s properties, the above approaches (i.e. [101, 103]) represent the system properties graphically. But, they still face the same problem of the property patterns (i.e. the system needs to be captured as a state-based model or an event-based model). In addition, they are still at the same level of abstraction of the formal notations (i.e. they do not provide a high level representation of the system properties).

---

**Figure 3-6:** Graphical representations of the formal notations to specifying a system’s properties

---

![Graphical notation for representing a real time system’s properties](image)

![A timeline for the property call waiting](image)

![The graphical elements of the property sequence chart](image)
**Property Sequence Chart** (PSC) is an extension to the UML sequence diagram to capture a software system’s properties [104-105]. The PSC graphical elements that can be used to specify a system property are shown in Figure 3-6.C. First, the property specified by these elements has a set of components (a component is captured as a rectangle with vertical dashed line as shown in Figure 3-6.C) that interact with each other by three types of messages (a message is captured as a horizontal line): fail, required, and regular. The fail message should never be exchanged between two components, while the required message must be exchanged when the message’s pre-condition is met. A message that is optionally exchanged between two components is called a regular message (see Figure 3-6.C). Second, the property has a timeline that specifies the order in which the messages are exchanged and it is from top to down. Third, the messages specified into the property can be grouped by a number of operators such as parallel (i.e. the messages are executed at the same time), strict (i.e. the messages should be executed in a specified order), loop (i.e. the messages are repeated a number of times), and alternative (i.e. a set of messages are selected to be executed). Finally, wanted and unwanted chains of messages can be specified by the property sequence chart (see Figure 3-6.C). An extension to the PSC notation to capture a system’s timed properties is also introduced by Zhang et al. [74].

Following the property sequence chart, a system’s properties are directly specified into its functionality model. Consequently, the system’s model complexity is increased and the system properties cannot be clearly captured.

### 3.3.2 Validating the Functional Requirements

In Section 3.2, we have discussed approaches that support the specification of a context-aware adaptive system’s requirements. These approaches do not focus on validating the functional requirements of the system. However, they extended requirements specification languages (e.g. Tropos [70] and goal models [71]), and there are approaches to validate a system’s functional requirements specified by such languages. In the following, we describe these approaches.

**Fuxman et al.** introduced an approach to validate a system’s requirements specified by the Tropos notation [23]. First, they introduced a language to formalize the requirements model which they call Formal Tropos (FT, for short). Following this language, a formal model for a system’s requirements is specified as two layers: outer and inner. The *outer* layer describes the elements of the requirements model and their attributes. The elements have five types: actors, goals, soft goals, tasks, and resources. The actors represent active entities that perform a set of actions to achieve the system goals (e.g. a student and a teacher). The goals state what need to be achieved by the actors (e.g. pass an exam). The soft goals represent the non-functional
requirements of the system (e.g. a student needs to maintain the integrity by studying the course to know correct answers in the e-exam). The tasks specify actions that need to be performed by the system actors (e.g. a teacher gives an exam) to achieve the system goals. The resources are information entities such as mark entity which provide the mark taken by a student in an exam. The inner layer defines a set of constraints into the elements of the outer layer. There are three types of constraints: invariant, creation, and fulfillment. The invariants specify constraints that should always hold during the system execution. An example invariant is that there is only one mark for a student in an exam. The creation and fulfillment constraints are conditions for creating and fulfilling a system goal. For example, the “get passing mark” goal is created when it is not fulfilled.

Second, to validate a system’s requirements specified using the Formal Tropos, Fuxman et al. developed a tool [23]. This tool transforms the functional requirements’ specification to a finite state machine that is acceptable by the NuSMV model checker [106]. The NuSMV is then used to check the requirements model against its constraints. Another approach is introduced by Kazhamiakin et al. to validate a system’s requirements specified by the Formal Tropos [107]. In this approach, the requirements model is transformed to a Promela code to enable its formal validation using the SPIN model checker [108].

Letier et al. introduced an approach to validate a system’s functional requirements specified by the KAOS goal model [30]. First, they introduced a formal representation of the goal model. An example formalized goal model for the mine pump system is shown in Figure 3-7. The main goal is to avoid sump overflow. This goal is decomposed into two sub-goals: maintain pump on when the water level is high, and maintain the water extraction when the pump is on. The goals also have their formal definitions. To achieve the mine pump system’s goals, a set of operations are specified. For example, the operation “start pump” is performed by the pump controller to switch the pump on. In addition, the operations have states from the domain before and after their execution. For example, the pump is off before performing the operation start pump, while it is on after executing this operation as shown in Figure 3-7. Furthermore, the operations have events that trigger their execution (i.e. ReqTrig), and events that should hold before and after the operations’ executions (i.e. ReqPre and ReqPost). For example, the start pump task is triggered when the water level is high and there no methane (see Figure 3-7).

Second, to validate a formalized goal model for a system’s requirements, they introduced a technique to derive a labeled transition system (LTS) and a set of system properties (specified by fluent liner temporal logic (FLTL) [109]) from the formalized goal model. The LTS model and the system properties are fed to the LTSA tool [58] for performing the validation. A similar
approach is introduced by Matoussi et al. [110] to validate a KAOS goal model, where the goal model is translated to Event-B specifications [111] to enable its validation.

![Figure 3-7: A formal model for the mine pump system’s requirements (Cf. [30])]

### 3.3.3 Checking the Adaptation Requirements Consistency

While a software system is in operation, it needs to be adapted in response to context changes. The system reactions to the context changes are specified using the adaptation requirements. To ensure that only valid actions are applied to the system in response to runtime context changes, the adaptation requirements need to be consistent. In general, the adaptation requirements are captured as condition-action rules, where the context changes trigger the evaluation of a set of conditions which in turn fire a set of actions (when the conditions are evaluated to true). Thus, in the following, we discuss approaches that have been introduced to check the consistency of a set of adaptation rules or a set of reaction rules in general.

**PobSAM** (Policy-based Self-Adaptive Model) is an approach to model a system’s adaptation requirements formally [112-113]. In this approach, the adaptation requirements are captured as a set of rules using the Rebeca language [114]. The adaptation model has two elements: self-adapt and enforce. The self-adapt element selects a system’s configuration in response to a context change. The enforce element specifies the actions that need to be performed into the system to
switch from current configuration to a new configuration. To ensure the system’s adaptation requirements consistency, a number of adaptation properties are specified manually using linear temporal logic (LTL) [73]. The properties include actions’ conflict and redundancy. Then, the Rebeca model is translated to Promela code [115]. Finally, the adaptation properties (captured as LTL formulas) and the Promela code are fed to the SPIN model checker [108] for performing the validation.

To use this approach, the software engineer needs to have a profound knowledge of formal methods to specify a system’s adaptation requirements and properties. This knowledge is usually lacking in the practice. Thus, the task of validating the system’s adaptation requirements becomes difficult.

Sama et al. introduced an approach to validating the adaptation requirements (specified as rules) of context-aware applications [25]. They analyse the adaptation rules to detect two types of faults: behavioural faults and context hazards. These faults may cause the firing of adaptation rules incorrectly or wanted rules are not triggered. The behaviour faults occur because of errors in specifying the rules’ conditions and the relationships between them. The context hazards are faults that arise because of inaccurate context information. To identify such faults, first, they construct an adaptation finite state machine (A-FSM) from the adaptation rules. An example finite state machine is a machine for a mobile phone application that captures the switching between different phone profiles in response to context changes. In this machine, the mobile phone switches from the general profile to the office profile when the user is at his office using “activate office” rule. Second, they use a set of algorithms to detect the behaviour faults and the context hazards. Each algorithm checks the constructed state machine against a property. When the property does not hold, it means a fault is detected. For example, one of the properties that need to be checked against the constructed state machine is “for each context situation, there at most one rule is fired”. Therefore, if two rules are fired in the same context situation, then the adaptation rules are not correctly designed. A tool has been also developed to automatically construct a state machine from a set of adaptation rules, and validate the rules to detect their faults (if any) [38].

Another approach is introduced by Xu et al. to detect and resolve context inconsistency that occur because of inaccurate context information [26, 116]. In their approach, they maintain an information repository that stores all the monitored context information. Every time new context information is received, an algorithm is used to detect its consistency with existing information. When this information is not consistent with existing ones, a policy is then used to either reject or accept this new context information.
The above two approaches (i.e. [25-26]) focus on detecting context consistency and ensuring that the correct adaptation rules are triggered. However, in the case of the context information is consistent and the correct rules are fired, the decided adaptation actions that need to be applied into the system may be inconsistent with each other. As such, the generated adaptation actions in response to context changes need to be also checked for consistency.

Also, there are a set of general approaches that are proposed for analysing condition-action rules (e.g. [117-119]) to detect unexpected behaviour of the rules (e.g. the rules may mutually trigger one another that leads to unexpected rules execution). However, these approaches do not consider specific properties of context-aware adaptive systems such as context consistency and the consistency of the generated adaptation actions in response to context changes.

3.3.4 Summary

Table 3-2 summarizes the approaches that have been introduced to specify a system’s properties and to validate a system’s functional or adaptation requirements. In the following, we discuss their limitations.

1- Specifying the System Properties: To specify the properties of a context-aware adaptive system, a number of approaches have been introduced. These approaches still have a number of limitations including difficulty in expressing the properties using the formal notations (e.g. [40, 73]), modelling the properties at a low level of abstraction (e.g. [101, 103]), and intertwining the system properties with the system functionality model (e.g. [104]). These limitations make the task of specifying the system’s properties difficult and error prone. In addition, the system properties can be local or global properties. The local properties are properties that need to only hold in specific context situations. The global properties should hold during the entire system execution (i.e. independent of the context situations). But, existing approaches only support the specification of the global properties without paying attention to specifying the local properties.

Table 3-2: A summary of approaches to specify a system’s properties and to validate its requirements

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying the System Properties</td>
<td>~ ~ ~ ~ ~ ~ ~ ~ ~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validating the Functional Requirements</td>
<td>~ ~ ~ ~ ~ ~ ~ ~ ~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validating the Adaptation Requirements</td>
<td>Context Consistency</td>
<td>+ - + -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Actions’ Validity</td>
<td>- ~ - -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

43
2- 
**Validating the Functional Requirements**: To validate the functional requirements of a context-aware adaptive system using existing approaches (e.g. [23, 30]), the system’s functional variants that correspond to the different context situations need to be enumerated and formally specified to enable their validation relative to a set of system properties. Thus, in a large scale system that has a high variability, a large number of functional variants need to be enumerated and specified which is difficult and an error prone task.

3- 
**Validating the Adaptation Requirements**: A number of approaches have been introduced to check the consistency of a system’s adaptation requirements. But, they still face a number of challenges. First, in some approaches (e.g. [113]), the adaptation requirements and properties to be checked against them need to be formally specified and then they cannot be easily adopted in practice. Second, a set of approaches (e.g. [25-26]) are able to check the context consistency and ensure that the adaptation rules are correctly fired in response to context changes. But, they do not check the consistency of adaptation actions that are generated in response to context changes (to ensure that only valid actions are applied to the system at runtime).

3.4 The System’s Design and Realization

Over the past decade a large number of approaches have been introduced to enable the design and realization of context-aware adaptive software systems. In the following, we discuss some of them. We classify these approaches based on which system aspect they are concerned with (i.e. the system structure or its behaviour), and their realization technique (i.e. by a middleware or using a model-driven approach).

3.4.1 Enabling the System’s Structure Adaptation

To enable the runtime adaptation of a system’s structure in response to context changes, a set of approaches have been introduced to design the system structure and to enable its realization.

**(A) Middleware/Framework-based Approaches**

**Rainbow** framework as shown in Figure 3-8 consists of two layers: architecture and system layers [16]. The architecture layer provides reusable mechanisms for monitoring changes to the system functionality and its resources using gauges, performing the analysis of these changes to initiate the adaptation process by a constraint evaluator, deciding the required adaptation actions by an adaptation engine, and effecting the needed adaptations to the running system through an adaptation executor. The system layer has the running system, probes and a resource discovery to monitor the system and its resources, and effectors to apply the decided adaptation actions to
the running system. The mapping between these two layers is performed by the translation layer as shown in Figure 3-8. In this approach, the system structure is represented by the ACME language [120]. The structure model is also maintained at runtime, so that it can be analysed to initiate the adaptation process in response to context changes. To capture the system’s adaptive behaviour, they introduced a language called Stitch [121]. This language captures the adaptive behaviour as a set of adaptation strategies in the form of decision trees. Each strategy is a set of tactics which in turn are sets of primitive operations (e.g. add and remove a service).

Following the Rainbow approach, an adaptive software system can be developed. However, in this approach, the context information is represented implicitly as a set of variables in the system model, and then the system modelling complexity is increased. In addition, specifying the system’s adaptive behaviour manually using the Stitch language is problematic in a large scale system, where a large number of system variations need to be captured at a low level of abstraction in a textual representation.

Realizing self-adaptive systems has many challenges that are posed by their complexity and adaptive behaviours, and then StarMX framework is proposed to ease the task of realizing such type of systems based on Java [76]. The framework is based on a rule engine to analyse context changes and to decide adaptation actions in response to the changes, and the Java Management Extensions (JMX) technology for sensing the context and effecting the decided actions. StarMX architecture has two main elements: execution engine and a set of services. First, the execution engine contains the self-adaptive operations (i.e. sensing, analysing, deciding, and effecting) as processes. Each process (e.g. effecting) is linked with specific system elements that provide the
needed functions (e.g. specific effectors that are created by the system developer) via anchor objects. Second, the set of services enable the execution of the adaptation operations. This set includes a lookup service to access anchor objects, an activation mechanism for triggering the adaptation process, and a caching service for improving the performance by holding references to the anchor objects.

Andrade et al. proposed a platform that supports the development of an adaptive system by separating the system’s adaptation logic from its core functionality [77]. The platform has three main modules: environment, adaptation, and redeployment. The environment module provides interfaces that need to be implemented to monitor specific environment entities. The adaptation module contains a set of condition-action rules in form of a tree that specifies a set of adaptation actions to be performed into the system in response to context (environment) changes (detected by the environment module). Each rule contains two parts: an adaptation condition as a Boolean expression based on a context attribute(s), and actions that represent the required architectural changes. The redeployment module is responsible for applying the required adaptation actions to the running system. It contains two methods: “handle adaptation” to perform pre-deployment operations to ensure the safe application of the adaptation actions, and “perform deployment” to apply the required adaptations.

The StarMX framework [76] and the approach introduced by Andrade et al. [77] support the design and realization of a software system’s adaptation logic. But, they did not consider how to design and realize the system functionality, the context information needed by this functionality, and the relationships between the system functionality and its adaptation logic.

Sykes et al. developed an approach to building self-adaptive systems [122-123]. They use labeled transition systems (LTS) [96] to capture a system’s domain model. This model captures the system states and environment (context) changes that move the system from one state to another by performing a set of adaptation actions. At runtime, in response to a context change, the domain model is used to derive a set of adaptation actions. These actions are then used by an architectural manager to generate a system’s configuration to keep achieving the system goals. The manager is aware of the available system components. Thus, it selects a set of components that are able to perform the generated actions, and composes them to form a new configuration. To ensure the correctness of this configuration, an Alloy representation of the new configuration is derived and validated [124]. In this approach, the context information is embedded into the system’s domain model. As such, the system modelling complexity is increased. In addition, the LTS model of the system is built based on the system states and the environment changes that trigger the switching between these states. Therefore, building an LTS model for a large scale
system is difficult and an error prone task, where a large number of system states and transitions between them (in response to environment changes) need to be captured.

**Folch et al.** introduced a technique that uses an architectural model for achieving runtime adaptability of a system [7]. The architecture model consists of the system’s basic components, and each component has a set of alternative variants. The variants have utility functions that are evaluated in response to a context change to specify which component variant suits the context change. The variants also have properties that need to be satisfied to use them. For example, a database (db) component may have two variants: caching db and basic db. These two variants have two different memory requirements and response times in different contexts. At runtime, in response to a context change, the system architecture variants are evaluated to select a variant that has the highest utility. Then, the MADAM middleware [125-126] is used for adapting the system from the current variant to the selected variant by applying a set of adaptation actions.

To specify a system’s adaptive behaviour using this approach, the software engineer should design a set of utility functions. When the system has high variability, the design of these utility functions is problematic, where complex functions need to be specified. In addition, annotating the system’s functionality model with a number of properties to specify its adaptation increases the complexity of the system’s design model.

**CASA** is a contract-based framework that enables runtime adaptation of a software system (an application) in response to changes into its resources (e.g. limited resources, resources with different reliability, or new resources are added) [127-129]. First, the system is designed to support the runtime adaptation by having a set of components that are active (i.e. running), and another set of passive components that are used for the system’s re-configuration (see Figure 3-9.A). Second, the system has a contract that specifies the system as a set of zones. Each zone has a set of configurations and their resource requirements to provide the system functions with a specific quality level (see Figure 3-9.B). As such, the system can adapt form one configuration to another in the same zone based on the available resources while maintaining the system’s quality requirements. Third, the framework has two elements: resource manager and contract enforcement system. The resources manager is responsible for maintaining the system resources status. The contract enforcement system gets the status of the system resources, and adapts the system based on the available resources by selecting a proper configuration.

Following the CASA framework [128-129], a large number of system configurations and the transitions between them need to be specified manually by the software engineer to consider the high variability of a context-aware adaptive software system. In addition, this framework does
not capture the context information and its relationships with the system explicitly. Therefore, designing and realizing the system by this framework is a difficult task.

**Figure 3-9:** Designing an adaptive system (application) using the CASA framework (Cf. [127])

**ROAD** is a framework that supports the development of adaptive software systems [45, 62, 130]. In this framework, the system is specified as two layers: organization and player layers. The organization layer specifies the system’s functional elements as roles, and the connections between the elements as contracts. The roles are abstract definitions of functions that need to be provided by the system while a set of players (i.e. the player layer) are responsible for providing the system functionality by playing the roles. The contracts specify the interactions between the system roles. The organization layer also has an organizer role that is responsible for adapting the system in response to runtime context changes by changing the system roles, contracts, and roles’ players binding. The ROAD framework has a runtime environment [47], where a system model that follows the ROAD model can be deployed to this environment to enable the system execution and runtime adaptation.

The ROAD framework enables the runtime adaptation of a system’s structure. However, it does not provide a mechanism to specify and realize the organizer player that is responsible for deciding the adaptation actions that need to be performed into the system in response to context changes. In addition, the framework considers the context model implicitly, and then the task of managing the context model and its relationships with the system becomes difficult.

(B) **Model-driven Approaches**

To adapt a software system while it is in operation, a runtime model of the system needs to be maintained. The runtime model’s elements are corresponding to the running entities (classes) of the system. Therefore, the runtime model is at a low level of abstraction, and then the task of managing the system is difficult. To ease this task, **Vogel and Giese** introduced an approach that transforms the system’s runtime model to high level models that capture different concerns.
of the system [131]. The architecture for the approach has a set of sensors to monitor the system and update the system’s runtime model, and a set of effectors to apply changes in the runtime model to the managed system using a set of pre-specified adaptation actions [132]. It also has a transformation engine that transforms the low level runtime model to a set of models at a higher level of abstraction, and reflects the changes in the high level models to the runtime model using Triple Graph Grammar (TGG) [133]. The high level models (which easier to be manipulated by the engineer) can be used by different adaptation managers to decide the system’s adaptation in response to context changes. This approach provides a high level representation of the running system to ease the task of its management. However, it does not capture the system environment and how its changes can trigger the system adaptation. It also does not provide a mechanism to specify the analysing and decision making operations of the adaptation manager.

Morin et al. introduced an approach to handle the exponential growth in the number of a system’s configurations that are derived from its variability [28, 134-135]. They combine model driven and aspect oriented approaches to cope with the complexity of adaptive systems. The conceptual model for this approach is divided into two phases: design time and runtime. The design time phase is concerned with specifying a system’s model as a base model (i.e. the core elements) and variant models (i.e. elements that can be waved into the base model in response to runtime context changes). An adaptation model is also specified as a set of adaptation rules that keep track of context changes and select variants that need to be composed with the base model to cope with the context changes. At runtime, a reasoning framework senses the context changes and fires the adaptation rules to identify variants that need to be composed with the base model. Then, a model composer waves the selected variants to the base model to have a configuration that suits current context situation. The configuration is then validated against a set of invariants using Kermeta [136]. When the generated configuration is valid, the system switches to this configuration by applying a set of adaptation actions to the running system.

The approach introduced by Morin et al. handles the expositional growth in the number of a system’s configurations. However, the context information and its relationships with the system functionality and management are not explicitly represented, and then the system model cannot be clearly captured and easily manipulated at runtime. In addition, the validation of a system’s configuration is performed at runtime which causes an overhead to the running system, and then this runtime overhead needs to be assessed to ensure that the system performance is not affected by the validation process.

MUSIC [34, 137] is a component-based middleware that optimizes a system’s overall utility in response to environment changes. The middleware has two parts: context and adaptation. The
context middleware is responsible for sensing context information, detecting context changes, and notifying the context changes to the adaptation middleware. The adaptation middleware decides adaptation actions to cope with the context changes. It has an adaptation controller that coordinates the adaptation by calling an adaptation reasoner to select a system’s configuration that suits current context. The selected configuration is then given to a configuration planner to generate an adaptation script which is performed into the system by a configuration executor. The MUSIC middleware also has a kernel to perform the re-configuration of a running system by creating, deleting, connecting, and disconnecting the system components.

To enable the design of a system by the MUSIC approach, they proposed a quality of service (QoS) model that describes the system composition together with the relevant QoS dimensions and how they are affected when the system is adapted from one configuration to another. This model is used for generating the system implementations. It also used at runtime for selecting a new system’s configuration that has the best utility and copes with a context change. They also introduced an ontology-based model to capture the context information that affects the system operations [138].

The MUSIC approach uses utility functions to capture the system adaptive behaviour, where a utility function is defined for each system component to specify in which context situation the component can be used. However, designing utility functions is problematic for large systems that have complex adaptive behaviours, where complex utility functions need to be specified.

3.4.2 Enabling the System’s Behaviour Adaptation

At runtime, a system’s behaviour (i.e. how system elements interact with each other to provide the system functionality) needs to be adapted in response to context changes. In recent years, a number of model-driven approaches have been introduced to design the system behaviour and to enable its realization. In the following, we describe and analyse some of them.

PLASTIC approach aims to support the adaptation of a web service in response to changes in the service context or in its non-functional requirements [78, 139]. A service model designed by this approach contains functional variants of the service. The service adaptive behaviour is then captured by annotating the service variants, where each variant is annotated by his resource demand, required context status, and quality of service (e.g. performance, reliability, etc.). Once a service model is created, the service implementations that suit different context situations and qualities can be generated from this model automatically. In addition, the service model can be transformed to a formal model for the validation. To execute the generated implementations, the Chameleon framework is used [78]. When a service need to be deployed to this framework, the
service context and required quality of service are acquired and a service implementation that suits them is selected and deployed.

In this approach, the service (system) adaptation is only limit to its deployment (discovery) time. In addition, to specify the service’s adaptive behaviour, the service variants need to be enumerated and then annotated with the situations that they can work in. Therefore, the task of designing the service variants is difficult and error prone (practically in complex services).

**Serendip** is an extension to the ROAD framework to enable the specification and realization of a system’s behaviour [63, 65, 140]. In Serendip, the system behaviour is specified as event-based processes. These processes have start events to specify when the processes are initiated, end events to define the processes termination, and a set of behaviour units. The behaviour units have a set of tasks to achieve the system goals. These tasks are defined based on the interactions that can be exchanged between the system roles. The tasks also have pre-conditions to enable their execution and post-conditions that are generated upon the tasks completion. To validate the system behaviour, a set of constraints are specified on the processes as computational tree logic (CTL) formulas. Then, the specified processes are transformed to Petri nets, so that they can be checked against the CTL formulas. The Serendip framework also has an engine to enable the processes’ enactment and their runtime adaptation.

The Serendip framework extends the ROAD framework to enable the runtime adaptation of a system’s behaviour. However, it still does not provide a mechanism to specify and realize the organizer player to enable the automatic system adaptation in response to anticipated context changes. In addition, the context-awareness aspect of the system is not explicitly specified and realized in this framework.

**Sheng et al.** introduced an approach to support the development of context-aware services. Their development environment (ContextServ) [13] relies on a meta-model called ContextUML [141-142]. This meta-model is based on four main concepts: context, context sources, service modeling, and context awareness. First, the context represents a web service’s environment and it can be an atomic context (i.e. single context attribute), or a composite context (i.e. aggregation of multiple context attributes). Second, the context sources specify context providers that are needed by a service and the switching between these providers based on the provided context quality. Third, the service modeling represents a service’s operations and their input/output messages. Fourth, the context awareness specifies a service’s reactions to context changes, and it is classified into two types: context binding and context triggering. The context binding means that the service needs a context attribute as an input parameter. The context triggering specifies
adaptation actions in response to runtime context changes. A service model designed using this meta-model can be also transformed to an executable service automatically.

The approach introduced by Sheng et al. only considers the development of single services without paying attention to the development of composite services. In addition, it only supports the adaptation of a service’s output parameter.

Zhang and Cheng introduced an approach to model an adaptive system formally [27]. In their approach, the system’s adaptive behaviour is separated from its no-adaptive (functional) behaviour. This separation makes the system models easier to be specified and validated. The system’s non-adaptive behaviour is a set of functional behaviours that the system needs to have to suit different context situations, while the adaptive behaviour is the switching between these functional behaviours. For example, an adaptive audio streaming system (described in [143]) can have two functional behaviours for the audio sender to suit two context situations: low rate for data loss, and high rate for data loss. The adaptive behaviour of this system is then modelled as the transitions between the two behaviours in response to changes in the data loss rate. They used Petri nets to model the system’s functional and adaptive behaviours. The Renew tool [144] is also used for the visual validation of the system behaviours by playing the token games, and for generating the Java implementation of the system from the Petri net models.

In a large scale system, the number of the system’s functional behaviours and the transitions between them is large. In addition, the software engineer needs to have a profound knowledge of formal methods to specify these behaviours using Petri nets. As such, designing and realizing an adaptive system using this approach is difficult and an error prone task.

Apto is an approach to developing adaptive service-based processes [75]. To develop an adaptive process by this approach, first, a model for the process is designed following the Apto meta-model that captures the process as four aspects: context model, linkage model, evolution model, and process model. The context model represents environment entities that trigger the process adaptation. It also has a number of constraints that represent context situations that need reactions from the process [145]. The evolution model specifies the possible changes (as evolution fragments) that can be performed into the process such as add flow, change flow, etc. A set of evolution fragments forms an evolution strategy. The linkage model defines which evolution strategy need to be executed in response to a context change (specified as a context constraint on the context model). The process model represents the service-based process that needs to be adapted. Second, when the process needs to be deployed, the context is monitored and the context constraints are evaluated based on the sensed context. The constraints that are
evaluated to true fires their linked evolution strategies. When these strategies are applied to the process model, a customized process model is generated and deployed to suit current context. Using this approach, the process adaptation is only limited to its deployment time, where once the process is deployed it cannot be changed anymore.

**PerCAS** is an approach to enable the switching between different behaviours of a service based on a user’s personalized adaptation logic [146]. A service design following this approach consists of a base functionality model (BFM), a model for the personalized context-awareness logic (PCLM), a personalization mechanism, and a weave mechanism. The BFM specifies the service’s basic functionality. The PCLM represents the context-awareness logic for a user as set of adaptation rules. The personalization mechanism switches between different PCLMs to suit individual users. The weave mechanism integrates the BFM and a PCLM to suit a user using an aspect oriented approach [147]. To enable the execution of the BFM and the PCLM models of a service, they are transformed to a BPEL process [148] and Drools rules [149]. Following this approach, in response to a change in the a user’s context-awareness logic, a new service need to be generated and deployed by integrating the base functionality model of the service with the changed logic where a deployed service using this approach is not changeable at runtime.

### 3.4.3 Summary

As discussed above and summarized in Table 3-3, a number of approaches have been proposed for designing and realizing context-aware adaptive systems that have the ability to be changed at runtime. However, they still have a number of limitations.

**1- System Functionality:** In some approaches (e.g. [27-28, 127]), a system’s functionality is designed as “N” functional models for “N” configurations (behaviours) to suit different context situations. As such, in a large scale software system, a large number of configurations need to be specified manually which is a difficult (complex) task. In addition, existing approaches focus on designing either the system’s structure (e.g. [7, 16, 34]) or its behaviour (e.g. [75, 78]). But, the system structure and behaviour need be modelled in a coherent manner to clearly capture the system elements and the sequence of interactions between them to achieve the system goals. Furthermore, in some approaches (e.g. [28, 76-77, 122, 131]), relationships between the system elements are not explicitly represented, and then the elements and their relationships cannot be clearly captured.

**2- Context Model:** Most of existing approaches do not represent the context information explicitly (see Table 3-3), where the context information is taken into account implicitly during the system implementation. Some approaches (e.g. [7, 77]) represent the context information
explicitly at the design time. But, the relationships between the context changes and the system functionality and management are not explicitly represented. As such, the effect of the context changes on the system cannot be clearly and easily captured.

3- **Adaptive Behaviour**: Existing approaches use state-based models (e.g. [27, 122]), utility functions (e.g. [7, 34]), or condition-action rules (e.g. [16, 77]) to capture a system’s adaptive behaviour. However, specifying the adaptive behaviour using these techniques is a complex task in the case that there are a large number of system reactions to context changes and the adaptive behaviour needs to be specified at a low level of abstraction. In addition, in some approaches (e.g. [75, 78, 128]), the system’s adaptive behaviour is intertwined with its functionality model and as such the system modelling complexity is increased.

**Table 3-3**: A summary of approaches to design and realize context-aware adaptive systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Design</strong></td>
<td>System Functionality</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>Context Model</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>I</td>
<td>E</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>Adaptive Behaviour</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>T</td>
<td>U</td>
<td>C</td>
<td>M</td>
<td>A</td>
<td>U</td>
<td>C</td>
<td>M</td>
<td>A</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>System Qualities</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Design Validation</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Managing the Complexity</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td><strong>System Realization</strong></td>
<td>Realization</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>Change Time</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>P</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>


4- **System Qualities**: Two types of quality need to be captured during the design of a context-aware adaptive system: functional and context. But, exiting approaches (e.g. [62, 78, 128]) only support the specification of the system’s functional qualities with little attention to specifying its context qualities and the effect of changes to these qualities on the system.

5- **System Model Constraints**: At runtime, an adaptive system switches from one state to another in response to context changes. While the system adapts, a number of local and global constraints need to hold to ensure that the new state (i.e. the system’s structure and behaviour) is consistent/valid. But, existing approaches (e.g. [63, 78, 122]) only support the specification of the global constraints. These constraints are also specified at a low level of abstraction.
6- **Design Validation:** To validate the design of an adaptive system by existing approaches (e.g. [27-28, 63]), the software engineer needs to have a profound knowledge of formal methods to specify the system formally. Also, a large number of system variants need to be specified to take the high variability of the system into account. Therefore, it is difficult to use the existing approaches to validate context-aware adaptive systems.

7- **Managing Complexity:** To cope with the complexity introduced by the high variability of a context-aware adaptive system, the system aspects (i.e. functionality, context, and adaptive behaviour) and their relationships need to be explicitly represented. But, existing approaches do not capture these three aspects and their relationships explicitly, and consequently the task of designing the system becomes complex. In addition, to design a large scale system, it needs to be modelled at different levels of abstraction, where the system model is simple at a high level while its details are included in the lower layers. However, some approaches (e.g. [75-76, 146]) only support the system modelling at lower levels of abstraction. As such, they cannot be easily used for modelling large scale context-aware adaptive systems.

8- **Design Synthesis:** To reduce the effort required for designing a system and to maintain the causal connection between the system requirements and design, the system design model needs to be synthesized from its requirements [36]. But, in existing approaches, the software engineer is responsible for translating the system requirements to a design model manually. Therefore, the causal connection between the system requirements and its design is maintained manually, which is a difficult task when the system is large. In general, there are approaches that support the synthesis of a system’s functional behaviour model from its requirements (e.g. [37, 150-151]). But, they do not support the synthesis of the system’s structure, context, and adaptation models from their requirements.

9- **System Realization:** Existing approaches provide either a framework/middleware (e.g. [16, 62, 76]) that can be extended to realize a system or a model-driven technique (e.g. [34, 75, 141]) where the system implementation is generated from its design. Context-aware adaptive systems are highly dynamic and have different aspects that are highly connected to each other. Thus, a model-driven technique is promising to realize these systems. However, the existing approaches generate executable systems only from models that are specified at lower levels of abstraction.

10- **Adaptation Time:** The system needs to be adapted to cope with context changes. Existing approaches either sense the context and adapt the system at the deployment time (e.g. [75, 78]), or adapt the system at runtime in response to the context changes (e.g. [77, 131]). The system is deployed into an environment that is highly dynamic. As such, approaches that adapt the system at deployment time only are not suitable for realizing context-aware adaptive systems.
System Flexibility: A software system’s aspects (i.e. functionality, context, and adaptive behaviour) need to be adapted at runtime in response to anticipated and unanticipated changes. But, the runtime changes in most of existing approaches (e.g. [7, 16, 28, 76, 122]) are limited to adaptations that are specified at the design time to cope with the anticipated context changes. As such, a system developed using these approaches cannot be changed at runtime to cope with the unanticipated changes. To cope with the unanticipated changes, a number of techniques have been introduced (e.g. [34, 62]). But, the context information needed by a context-aware adaptive system (to operate effectively or to trigger its adaptation) and its relationships with the system’s functionality and management are not explicitly realized. As such, it becomes difficult to evolve the system developed in these approaches when the unanticipated changes occur at runtime.

3.5 The System Evolution to Cope with Unanticipated Changes

During its lifetime, a context-aware adaptive system needs to evolve to cope with changes that were not anticipated during the development time. To evolve the system, changes to the system requirements need to be captured and reflected to the system’s design model. These changes are then validated and realized to the running system. In the following, we describe approaches that have been introduced to support these engineering tasks.

3.5.1 Specifying the Changes to the System Requirements and Design

While a context-aware adaptive software system is in operation, new functional and adaptation requirements may become needed or existing requirements need to be changed. To cope with such changes, the system requirements specification and design model need to be modified.

Cote et al. introduced an approach to enable the evolution of a software system architecture in response to requirement changes [29, 48]. At the development time, the system requirements are specified using the problem frames in two steps. The first step represents the system’s main problem, while the second step decomposes the main problem into sub-problems so that it can be easily evolved. The sub-problems are then fitted to categories so that corresponding design patterns can be chosen for each sub-problem. When the requirements evolve, the two steps are repeated to consider the changes in the requirements. First, the main problem specification is changed by adding new elements to this specification or by modifying its elements. Second, the sub-problems are changed to reflect the changes in the main problem by introducing new sub-problems or by modifying existing ones. Then, the new and/or modified sub-problems are fitted to categories to select appropriate design patterns for the new sub-problems or to adjust the old system’s design to consider changes in the existing sub-problems. This approach targets general software systems. In addition, it gives a general methodology to evolve the system requirements.
specification and its design model without providing a guideline or a set of change patterns to ease the system evolution task.

Gomaa et al. introduced a set of re-configuration patterns that specify possible changes to a software system [79, 152]. These patterns are defined based on design patterns that describe a system’s components and their interconnections such as master/slave, client/server, etc. For each design pattern, a re-configuration pattern is identified to specify the operational behaviour of the design pattern’s elements to execute a set of re-configuration commands (e.g. replacing a client with another in the client/server model). The patterns are also formalized using a pattern template introduced by Ramirez and Cheng [153-154].

LeGoaer et al. introduced the term evolution style [50, 155]. An evolution style specifies a change that can be performed into a system’s structure such as “add an element”. An example evolution style is “adding a sub-class to a system’s structure captured using the class diagram”. This style has a name “AddSubClass”, a goal “add a new class as a specialization of an existing class”, a domain which specifies the language that this style is defined for (e.g. Famix [156]), a header that specifies the pre and post conditions of the evolution style, and competence that defines the sequence of changes that need to be executed when the style is applied.

Similarly, Ahmad et al. have proposed the concept of change pattern [157]. The change pattern specifies a common sequence of changes to a software system’s structure. They also have introduced an approach to identify change patterns from the change logs of a number of software systems.

These above approaches (i.e. [79, 155, 157]) provide the bases for specifying and organizing a set of change patterns (or evolution styles) to support a software system’s evolution. However, they focus on general software systems, where they only specify possible changes to a system’s functionality. Therefore, to identify change patterns for context-aware adaptive systems, these approaches need to be extended to capture the system’s adaptation and context aspects and to specify possible changes to the system’s different aspects in response to unanticipated changes.

3.5.2 Validating the Changes

After changing a context-aware adaptive system’s requirements and design models in response to unanticipated changes, the models need to be validated to ensure that they are still valid. The validation approaches that proposed to validate the initial models of the system’s requirements (e.g. [23, 30]) and its design (e.g. [27, 123]) can be used to validate these models again when changes are introduced to them. But, validating the whole system models is a time consuming
task, and the system may change a number of times in small duration. Therefore, it is inefficient approach to re-validate the whole range of system models.

An approach is introduced by Morin et al. to validate a system’s configurations at runtime [28], where a configuration is only validated when it is needed. This approach reduces the time required for validating the system configurations when changes are introduced to them. But, the late validation may identify problems that need to be resolved, and then the system needs to be stopped so that the configuration is corrected and then used. Thus, this approach is not suitable for context-aware systems that need to evolve without shutdown.

A number of approaches are introduced to validate a system incrementally (e.g. [67, 158-159]). In such approaches, the full system model is not explored, where the model is divided to smaller sub-models and the validation is performed into these sub-models incrementally. These approaches are faster than the traditional approaches, where the full state space of the system model is not explored during the validation of a system property. But, using them for validating the evolved models of a context-aware software system is still a time consuming task, where all system configurations and/or behaviours need to be validated.

3.5.3 Realizing the Changes

To increase customers’ satisfaction, changes in a system’s design model should be realized to the system at runtime. In the following, we describe approaches that are introduced to support this engineering task. We classify them into two types: code-based and script-based approaches. In the code-based approaches, the changes are performed into the system’s source code, and then a tool applies these changes to the system at runtime. The script-based approaches enable the engineer to write an adaptation script that can be directly executed into the running system to incorporate new elements or to modify existing system elements.

(A) Code-based Approaches

To dynamically update a single threaded “C” program, the Ginseng approach is introduced [68]. It has three main elements: a compiler, a runtime environment, and a patch generator as shown in Figure 3-10. Using the compiler an updateable program can be generated which can run on the runtime environment (e.g. version “V0” shown in Figure 3-10). To update a running program, the program’s source code is changed (e.g. version “V1”). Then, the patch generator compares the version of the running program with its new version and generates a patch script. Finally, the compiler compiles this script into a dynamic patch that can be loaded by the runtime environment to update the running program. To enable the safe update, this approach constrains
the runtime updates to certain points when none of changed elements are in use. This process can be repeated a number of times in response to unanticipated changes.

**Figure 3-10:** Developing and updating a software system using the Ginseng approach (Cf. [68])

To support runtime update of multi-threaded systems written in “C” language, an approach was introduced which called POwerful Live Updating System (POLUS) [32, 160]. It achieves this by allowing the co-existence of the old and new states of a system and it synchronizes them together when the update happens. The POLUS approach is based on three elements: a patch constructor, a runtime library, and a patch injector. The patch constructor generates a patch that contains differences between the old and new versions of the system, and a code to maintain the state consistency among the system threads. The runtime library and the patch injector update the running system by injecting the generated patch to it.

**Java HotSwap** is an approach to allow programmers to swap currently executing classes with new ones [161]. First, a new version of a class is developed and compiled. Second, a GUI client is used for uploading the new class and the necessary code to perform the swap. Finally, the Java Virtual Machine (JVM) calls RedefineClasses() internally to replace the executing class with the new version. Java HotSwap only allows changing method bodies, and then there is no need for any state transfer where everything else in the updated class stays exactly the same.

Another approach to update Java programs is **JVolve** [31, 162]. It has an Update Preparation Tool (UPT) that generates mappings between different versions of a Java application. In order to perform the runtime update, the tool stops the running application and uses these mappings to replace its old classes with the modified ones. Then, state transformers are applied so that the application returns to a valid state when re-activated. To identify safe points for updating the running application, they restrict which methods are allowed to reside on the call stack when an update will take place.
The above approaches to runtime update of a system are tied to a particular programming language (e.g. Java [31] and C [68]). Therefore, using these approaches to realize changes of a context-aware adaptive system faces a number of challenges. First, in a large scale system, several changes need to be made in different places of the system code, and then tracking the changes and their effect on each other is a difficult task for the system developers. Second, the patch script used for updating the system is generated after implementing the changes. As such, if there are violations detected by the script generator, the developer is notified to resolve them. This process may take long time where several rounds are made between the programming and script generation phases (i.e. a time consuming task). Finally, these approaches are working at the code level and there is no explicit connection between the system model and the code. Thus, it is difficult for the stakeholders to understand the changes at the code level.

(B) Script-based Approaches

Oreizy et al. introduced an architecture-based approach for runtime system evolution [33]. In this approach, an instance of the system architecture is maintained at runtime. The system adaptation in response to unanticipated changes is then specified at the architecture level using a set of actions such as add, remove, or replace a component. They have used the ArchStudio tool [163] for realizing changes to the system (that is written using Java-C2 class framework [164]). The tool enables the software engineer to write a script that specifies the required changes to the system architecture. The script is then used by the tool to evolve the running system.

Andrade et al. introduced an approach to cope with unanticipated changes to the adaptation logic of a system [77]. They separate the system’s adaptation logic from its functionality, and represent the adaptation logic as condition-action rules that specify adaptation actions that need to be performed into the system in response to context changes. The rules are constructed as a component-based system that can be changed at runtime. To change the system adaptation rules, a script that contains the needed changes to the rules is written. The script is then executed to evolve the adaptation rules at runtime.

Similar to the above two approaches, there are a number of approaches that enable runtime evolution of a system’s functionality and its adaptation logic by writing and executing scripts that have changes that need to be applied to the system while it is in operation (e.g. MUSIC [34] and ROAD [47] frameworks).

The script-based approaches support runtime changes to the system’s adaptation logic and its functionality to cope with unanticipated changes. However, the software engineer is responsible for writing scripts that contains changes that are then applied to the system at runtime. He also
needs to have a good knowledge about the artifacts of the running system to specify the required changes correctly. Thus, this task becomes tedious and error prone in large scale systems where complex adaptation scripts need to be written manually.

3.5.4 Summary

Table 3-4 summarizes the approaches that have been proposed to support the runtime evolution of context-aware adaptive systems. In the following, we discuss their limitations.

1- *Specifying the Changes*: A set of approaches have been introduced to support a system’s evolution from early stages such as specifying changes to the system requirements specification and design model. Using these approaches, the changes are specified in an ad-hoc manner (e.g. in [48]), or following a number of change patterns (e.g. in [79, 155]). However, they only focus on specifying changes to the system functionality, and they cannot be used to specify changes to the adaptation and context aspects of a context-aware adaptive system.

2- *Validating the Changes*: To validate the changes of a context-aware adaptive system’s requirements specification and design model in existing approaches (e.g. [23, 27, 67, 158]), all system variants need to be identified and validated, which is a time consuming task. Also, an approach is proposed to only validate a system variant when needed at runtime [28]. But, the late validation and detection of problems in a system variant triggers the need to stop the system to resolve the problems, and this is not acceptable in systems that should be available 24/7.

3- *Realize the Changes*: A set of approaches have been proposed to realize runtime changes to a running system. However, the runtime changes to the system are specified directly against the system’s source code (e.g. [160-161]), or specified textually as adaptation actions that need to be applied to the system (e.g. [33-34]). Thus, in a complex system, this task (i.e. specifying the runtime changes) is tedious and error prone. In addition, existing approaches do not support the incorporation of new context information into a context-aware adaptive system at runtime.
3.6 Challenges to Develop and Evolve Context-aware Adaptive Software Systems

In the previous section, we have discussed approaches that aimed to support the development and runtime evolution of a context-aware adaptive system. These approaches assist the software engineer in different engineering tasks such as specifying the system functional and adaptation requirements (Section 3.2), validating the system requirements (Section 3.3), designing and realizing the system (Section 3.4), and evolving the system in response to unanticipated changes (Section 3.5). However, developing and evolving context-aware adaptive systems using such approaches still faces a number of challenges as discussed above.

First, in existing approaches, the adaptation requirements of a system are intertwined with its functional requirements specification. In addition, the context information needed by the system (to operate effectively or to trigger its adaptation) is not explicitly represented. Furthermore, the system requirements are captured as goals which are usually difficult for the stakeholders to articulate. Thus, in Chapter 4, we present a novel approach to specify the system requirements as two sets of scenarios (i.e. functional and adaptation scenarios). The functional scenarios capture the system functionality, while the adaptation scenarios specify the system adaptation in response to context changes. In such scenarios, contexts and their relationships with the system functionality and adaptation are explicitly represented.

Second, existing approaches do not enable the consistency checking of a system’s adaptation requirements. In addition, to ensure the validity of the system’s functional variants, the variants need to be enumerated and specified manually, which is a difficult task (when the system has high variability). Thus, in Chapter 4, we introduce a technique for enumerating and generating the functional variants of a software system from its requirements specification. It also checks the consistency of the system’s adaptation requirements.

Third, the aspects of a context-aware adaptive software system (i.e. functionality, context, and management) and their relationships are not explicitly represented (designed) and realized in existing approaches, and then these aspects cannot be clearly captured and easily changed at runtime in response to anticipated and unanticipated changes. In addition, the software engineer is responsible for translating the system requirements to its design model, and then the causal connection between the system requirements and its design is maintained manually which is a difficult task when the number of requirements is large. As such, to ease the task of designing a context-aware adaptive system, we introduce an organization-based meta-model in Chapter 5. This meta-model captures the system’s three aspects and their relationships explicitly, so that
these aspects and their relationships can be clearly captured and easily manipulated. We also present a number of algorithms that enable the automatic synthesis of the system design from its requirements (specified as scenarios). Furthermore, we adopt the models@runtime concept to keep the system model alive at runtime, so that the different aspects of the system can be easily changed at runtime in response to anticipated and unanticipated changes.

Fourth, to cope with unanticipated changes, existing approaches specify the changes to a system’s requirements and its design in an ad-hoc manner or using patterns that only specify the changes to the system functionality. In addition, to validate the changes, the changed system variants need to be identified manually, which is a difficult task in a large scale software system. Furthermore, to realize the system changes, these changes need to be specified at a low level of abstraction, and as such it is an error prone and tedious activity. In Chapter 6, we introduce a set of evolution (change) patterns that assist the software engineer in evolving a context-aware adaptive system’s requirements specification and its design model. To validate the system’s changes, we also introduce a technique for identifying and validating the changes automatically and incrementally. Finally, to realize the system changes, the software engineer modifies the system scenarios, and then the scenarios changes are reflected to the system design (using the change patterns) and applied to the running system automatically.

In a nutshell, this thesis introduces a scenario-driven approach to assist the software engineer in specifying the requirements of a context-aware adaptive system, validating the requirements specification, synthesizing the system design model from its requirements, realizing the system based on its design model, and evolving the system to cope with unanticipated changes. These aspects of the approach are discussed in Chapters 4, 5, and 6 (see Figure 3-11).

![Figure 3-11: Developing and evolving context-aware adaptive software systems](image)
3.7 Summary

In this chapter, we have analysed existing approaches that assist the software engineer in the development and runtime evolution of context-aware adaptive systems. These approaches are analysed with respect to the set of requirements identified in Chapter 2, and classified into four groups relative to the engineering tasks supported by them. The approaches are described briefly and analysed individually to identify their limitations. In addition, the limitations of approaches in each group are summarised and a comparison between them is performed to identify research challenges that are still remain.

We have also discussed in brief how the identified research challenges are tackled in our scenario-driven approach to developing and evolving context-aware adaptive software systems.
Part II: The Approach
Chapter 4: Specification and Validation of the System Requirements

The success of a software system is measured by the degree to which the system meets its users’ needs (requirements). Therefore, specifying the system requirements is an important step during its development [165]. In addition, ensuring the validity of the requirements is an important task to avoid the high costs that are introduced when requirements’ errors are detected and fixed late at the software development life cycle [166]. For a context-aware adaptive software system, two types of requirements need to be specified: functional and adaptation [36]. The functional requirements capture the system’s core functionality, while the adaptation requirements define the system reactions to context changes. These requirements need to be also captured in a form that is easily understandable by the stakeholders. In addition, to ensure the validity of the system requirements, a large number of system variants (that correspond to different context situations) need to be validated [28].

As discussed in Sections 3.2 and 3.3, in existing approaches, the specification of adaptation requirements for a context-aware adaptive system is intertwined with its functional requirements (e.g. [19-21]). In addition, these approaches do not explicitly capture the context information and its relationships with the system’s functional and adaptation requirements. Furthermore, the system requirements are captured as a set of goals. But, it is often difficult for the stakeholders to articulate their needs as a number of goals [22]. Thus, it is difficult to use such approaches to specify the functional, contextual, and adaptation requirements of a context-aware adaptive software system. To validate the functional requirements of a context-aware adaptive system, the system variant specifications need to be manually enumerated and specified by the software engineer in the existing approaches (e.g. [23-24]). However, in complex systems that have high variability, the number of variants is large and the task of requirements validation becomes impractical. In addition, the existing approaches do not check the consistency of a software system’s adaptation requirements to ensure that only valid actions are applied to the system at runtime (e.g. [25-26]). Thus, in this chapter, we introduce a novel scenario-based approach to specifying and validating the varying requirements of context-aware adaptive systems.

First, the approach specifies the functional requirements of a system as a set of functional scenarios (a scenario is a sequence of events that is one possible pathway through a system’s behaviour [22]). In particular, we identify the environment context and its functional use in these scenarios. In addition, to capture the system’s adaptation requirements, we propose the concept of adaptation scenarios. These scenarios specify adaptations to the functional scenarios.
in response to runtime context changes. To specify these two types of scenarios, we extend the UML sequence diagram [167] with three types of scenario participants: functional, contextual, and management. The functional participants provide the system’s core functionality, while the management participants decide and perform adaptation actions on the functional system (or the functional scenarios) to cope with context changes. The contextual participants provide context information that is needed by the system functionality, or that can trigger its adaptation. These participants interact with each other by special types of messages. For example, a management message can be only exchanged between management and functional system participants. We also extend the UML sequence diagram to specify in a form similar to the scenarios the system properties that need to be preserved at runtime when the system adapts itself according to the adaptation scenarios.

Second, to validate a system’s variants, we use the system’s adaptation scenarios to generate adaptation scripts each of which is a set of adaptation actions that need to be applied into the system in response to runtime context changes. These scripts are also checked for consistency to identify errors including redundant, conflicting, and missing adaptation actions. The consistent scripts are then applied to the functional scenarios to generate possible variants of the functional requirements. Finally, the variants and the system properties are transformed to Petri nets [39] and computational tree logic (CTL) [40] respectively, before being fed into an existing model checker (i.e. the Romeo tool [41]) for the conformance check.

This chapter is organized as follows. Section 4.1 presents our approach to specifying the requirements and properties of a context-aware adaptive software system. Section 4.2 discusses a technique to validating the system’s functional variants against the expected properties, and to checking the consistency of the system’s adaptation requirements.

4.1 Specifying the System Requirements

While a context-aware adaptive system in operation, it switches between its variants to cope with context changes [88]. Thus, the system can be viewed as multiple functional variants that suit different context situations. These variants consist of common and variable parts [7]. The common part is shared between all system variants. The variable parts are variations introduced into the common part to suit the different situations as shown in Figure 4-1.A. At runtime, to switch between two system’s variants, adaptation actions corresponding to differences between the two variants are applied to the current variant, so that the system moves to the target variant.

---

1 The term variant is used throughout the chapter to describe a set of functional scenarios (requirements) that the system is able to support (provide) in a specific context situation.
An example switching between three system’s variants is shown in Figure 4-1.B. Consequently, to specify the system requirements, the system variants that suit different context situations need to be first specified (i.e. the functional requirements). Then, the switching between the variants in response to context changes is defined (i.e. the adaptation requirements).

**Figure 4-1:** Runtime adaptation in context-aware software systems

The high variability of a context-aware adaptive software system introduces an explosion of its variants [28]. Therefore, specifying this large number of system variants and the switching between them is complex and an error prone task. To tackle this problem, in our approach, we represent the functional requirements of all system variants as one model (i.e. the “functionality model”) instead of “N” models for “N” variants as shown in Figure 4-2.A. We also introduce an “adaptation model” for specifying actions to be applied into the functionality model to suit different context situations. As such, using the functionality and adaptation models, first, the system’s variant specifications can be generated by identifying adaptation actions that need to be applied into the system in response to context changes. Then, these actions are applied to the functionality model to generate the system’s functional variants (see Figure 4-2.A).

**Figure 4-2:** Specifying a system’s variants and the switching between two of them in our approach
Second, to switch between two system’s variants in response to a context change at runtime, adaptation actions to cope with this change are identified by the adaptation model and used for generating a target variant. Then, differences between the current and target variants are computed. Finally, the actions corresponding to the differences are applied to the current variant, so that the system moves to the target variant that suit the new context situation as shown in Figure 4-2.B.

To specify the functional and adaptation requirements of a context-aware adaptive system as functionality and adaptation models (discussed above), we adopt a scenario-based approach because its simplicity and intuitive graphical representation facilitate stakeholder involvement in capturing the system requirements [22]. It also has a well-understood and widely accepted semantics [37]. Following the scenario-based approach, we capture the system functionality as a set of functional scenarios, and its adaptation requirements as a set of adaptation scenarios that specify adaptations to the functional scenarios in response to context changes. To facilitate the validation of the system’s requirements, the system properties that need to be preserved when the system adapts at runtime are also captured in a form similar to the scenarios.

4.1.1 Background: The UML Sequence Diagram

To specify a system’s requirements as a set of scenarios, a number of scenario-based notations have been introduced [37, 168]. In this thesis, we adopt and extend the widely accepted UML sequence diagram to specify a system’s functional and adaptation requirements, and the system properties that need to hold at runtime. Before introducing our extensions, we describe below the core elements of the UML sequence diagram.

Following the UML sequence diagram, a scenario consists of lifelines, messages, and interaction fragments (see the top part of Figure 4-3). The lifelines represent the scenario participants, while the messages capture the interactions between them. To capture the scenario flow, the sequence diagram has a set of interaction fragments including an interaction use, an occurrence specification, a state invariant, and a combined fragment. First, the “interaction use” specifies that a scenario can refer to (or use) another scenario. For example, in Figure 4-4, the scenario “FS1” has a reference to the “plan a route” scenario. To specify a high level view of the system, a scenario can be defined that has references to the different scenarios of the system and specifies their relationships as shown in Figure 4-4. Second, the “occurrence specification” concerns the intersection point between a participant and a message to define what events are sent or received by a participant. For example, the intersection point between the login message “FM1” and the user participant (see Figure 4-4) specifies that the user can request the login operation from the provider of the travel guide system.
Third, the “state invariant” is a runtime constraint on a lifeline, and the evaluation of this constraint to true means that the running sequence is a valid trace. For example, the sequence shown in Figure 4-4 is a valid trace if the “LoginSuccess” variable equals to “True” before the user is able to use the system services (e.g. use the route planning).

Figure 4-3: A meta-model for the extended UML sequence diagram to specify the functional scenarios

Figure 4-4: An example scenario that shows the user interactions with the travel guide provider
Fourth, the “combined fragment” is used to group a set of interactions and define their relationships (e.g. alternative, loop, etc.). Each combined fragment consists of an operator and operands (see Figure 4-3). The operator defines the type of the combined fragment (e.g. alternative), while the operands specify interactions that are grouped by the combined fragment operator. In addition, each operand can have a constraint as a Boolean expression that specifies when this interaction can be executed. An example combined fragment is the loop fragment. It is used to specify that an interaction or a number of interactions are repeated as long as the loop condition is true. For example, the interaction “FM1: Login” is repeated a number of times until the condition “LoginSuccess == False” becomes false as shown in Figure 4-4.

4.1.2 Specifying the System’s Functional Requirements

To capture a system’s functionality while taking into account the context information needed by this functionality as a set of scenarios, we have extended the UML sequence diagram by having two types of scenario participants (i.e. functional and contextual) that interact with each other through special types of message as shown in the bottom part of Figure 4-3.

(1) Functional Participants and their Messages: The functional participants are responsible for providing the system functionality. For example, the “attractions finder” participant is responsible for suggesting a set of attractions as shown in Figure 4-5. The messages between the functional participants are either requesting or providing a functional operation, and therefore called “functional messages”.

Figure 4-5: Sequence of interactions to suggest a number of attractions to the user

---

2 Types of messages that can be exchanged between a scenario’s participants depend on the types of the participants (e.g. a contextual message cannot be exchanged between two functional participants).
(2) **Contextual Participants and their Messages**: Context information is needed by a context-aware system to better carry out its functionality (e.g. considering the weather conditions in suggesting the attractions) [141]. As such, we make the *contextual* participants explicit in the functional scenarios to provide the context information (e.g. the weather information provider in Figure 4-5). The functional participants can be notified of context changes or they can request the context information on-the-fly. Consequently, we have two types of contextual messages *(notify and get context)* that can be exchanged between functional and contextual participants as shown in Figure 4-3. For example, the “attractions finder” can get the weather information from its provider (i.e. CM1: “get context” shown in Figure 4-5). To specify the system reactions to changes in the context information that is needed by its functionality, alternative fragments can be used where different interactions are executed based on the context status. For example, the interaction “FM2: ProvideIndoorAttractions” is executed when the weather condition is rainy, while “FM3: ProvideOutdoorAttractions” is executed otherwise as shown in Figure 4-5.

(3) **Functional and Context Qualities**: In specifying the functional requirements, there is a need to define some qualities about the system functionality (e.g. response time, reliability, etc.) [52] and the context information (e.g. accuracy, freshness, etc.) [53]. To specify such qualities, a number of extensions to the UML sequence diagram are introduced (e.g. Real-Time UML [169-170] and QML [171]). However, they only capture the system’s functional qualities as a set of constraints. As such, similar to these extensions, we extend the “state invariant” of the UML sequence diagram to capture the *functional* and *context* qualities of the system as constraints (see Figure 4-3). This extension enables the representation of quantifiable (measurable) quality attributes. Example quality requirements are: (1) *response time* of the route planning function “ProvideRoutes1” need to be less than 7 seconds, and (2) *freshness* of the traffic information should be up to the last two minutes as shown in Figure 4-6.

Example functional scenarios are “FS1”, “FS2”, and “FS3” shown in Figure 4-4, Figure 4-5, and Figure 4-6. The scenario “FS1” captures the tourist interactions with the overall travel guide system. He can login to the system. After login, he can use the system to find attractions and plan a route to visit a number of attractions. The scenario “FS2” describes the details of how the travel guide suggests attractions to the tourist based on the weather conditions (e.g. when the weather is rainy, only the indoor attractions are suggested). The tourist interactions with the route planning sub-system to plan a route are captured in the scenario “FS3”. First, the tourist requests a route. Then, the route planner provides a set of routes to him while taking the traffic information into account. Finally, he can select a suitable route and start his journey.
In naming a scenario’s elements, we give each element an identifier, so that it can be easily referenced by the adaptation scenarios (as described below). The identifier has two letters that represent the element type and an auto-generated number to differentiate between elements of the same type in the scenario. For example, the identifier “FM1” shown in Figure 4-6 means “this is the first functional message in the scenario”. The right part of Figure 4-4 presents the abbreviations of each element type and their meanings.

4.1.3 Specifying the Adaptation Requirements

Similar to capturing the functional requirements as a set of functional scenarios [37, 168], we propose the concept of adaptation scenarios to capture the adaptation requirements. Such scenarios describe adaptations to the functional scenarios (requirements) in response to context changes, and they are built in the same way that the functional scenarios are built. To define the adaptation scenarios, we extend the UML sequence diagram by introducing a management lifeline, a functional system lifeline, management messages, and adaptation fragments as shown in the bottom of Figure 4-7. The adaptation scenarios also have contextual participants.

(1) Contextual Participants: Each adaptation scenario has contextual participants to provide context information that triggers the system adaptation. For example, changes in the user’s selected features cause the travel guide system’s adaptation. Thus, we have a contextual lifeline “CL1” to make this information available (see Figure 4-8). In general, the context information
that triggers the system adaptation can be about the end users (e.g. the user’s selected features as shown in Figure 4-8), the system environment (e.g. the traffic information availability as shown in Figure 4-9), the non-functional attributes (e.g. the response time of the route planner), or the context quality (e.g. the traffic information freshness).

Figure 4-7: A meta-model for the extended UML sequence diagram to capture the adaptation scenarios

(2) Functional System Lifeline: In an adaptation scenario, the functional system participant represents the system’s functionality (i.e. the functional scenarios) that needs to be adapted in response to context changes (e.g. the travel guide system’s functionality “FL1” in Figure 4-8).

(3) Management Participant and its Messages: A management participant in an adaptation scenario is responsible for deciding and performing the system adaptation in response to context changes, i.e., the organizer participant “ML1” in Figure 4-8. It uses a number of management messages to specify adaptations that need to be applied into the functional scenarios. Example adaptations that need to be applied into the functional scenario “FS3” are: remove the message “CM1” and add the lifeline “CL1” as shown in Figure 4-9. In general, we have adaptation actions to add, remove, or change the scenario elements (i.e. lifelines, messages, and interaction fragments) as shown in Figure 4-7. In the following, we list possible adaptation actions that can be performed into the functional scenarios:

- \textit{AddLifeLine} (ScenarioId, LifeLineId)
- RemoveLifeLine (ScenarioId, LifeLineId)
- AddMessage (ScenarioId, MessageId)
- RemoveMessage (ScenarioId, MessageId)
- ChangeMessageParameters (ScenarioId, MessageId, NewParametersList)
- AddCombinedFragment (ScenarioId, FragmentId)
- RemoveCombinedFragment (ScenarioId, FragmentId)
- ChangeCombinedFragmentCondition (ScenarioId, FragmentId, NewCondition)
- AddMessageToCombinedFragment (ScenarioId, FragmentId, MessageId)
- RemoveMessageFromCombinedFragment (ScenarioId, FragmentId, MessageId)
- AddStateInvariant (ScenarioId, StateInvariantId)
- RemoveStateInvariant (ScenarioId, StateInvariantId)
- ChangeStateInvariant (ScenarioId, StateInvariantId, NewStateInvariant)
- AddSequenceReference (ScenarioId, SequenceReferenceId)
- RemoveSequenceReference (ScenarioId, SequenceReferenceId)
- ChangeSequenceReference (ScenarioId, SequenceReferenceId, NewSequenceName)

**Figure 4-8:** An adaptation scenario to cope with changes in the user selected features (services)

(4) *Adaptation Fragment:* To specify what to adapt in response to context changes, we introduce the “adaptation operator” (see Figure 4-7). This operator groups a set of management
messages (i.e. adaptation actions). We also extend the interaction constraint with the “context condition” (see Figure 4-7) to specify a contextual situation in which the system needs to adapt itself (e.g. the user wants to include the route planning service). We call the combined fragment that has the adaptation operator and the context condition the “adaptation fragment”.

An example adaptation scenario is “AS2” shown in Figure 4-9. This scenario contains two adaptation fragments: “AD1” and “AD2”. The fragment “AD1” specifies the required changes to the scenario “FS3: plan a route” when the traffic information is not available, i.e., removing the traffic information participant “CL1”, the message “CM1”, and the context quality “CQ1”. The fragment “AD2” has the reverse of these three actions that need to be executed when the traffic information becomes available. The execution of the two fragments is enabled by the message “CM1” (see Figure 4-9). Generally, in specifying the adaptation fragments, adaptation actions and their reverse need to be defined, so that a set of scenario elements only exist when a context situation hold while they are removed otherwise (as they are not needed).

![Figure 4-9: An adaptation scenario to cope with changes in the traffic information availability](image)

Using the functional and adaptation scenarios, the system variants that suit different context situations can be generated where in response to context changes, adaptation actions are decided by the adaptation scenarios. Such actions are then applied to the functional scenarios to generate the system variants that are able to cope with the context changes. Example two variants that are generated from the functional scenario “FS3” are presented in Figure 4-10. The two variants are used to cope with changes in the status of the traffic information availability, where the lifeline “CL1”, the message “CM1”, and the quality “CQ1” are removed when the traffic information is not available (i.e. variant 1) and added otherwise (i.e. variant 2).
4.1.4 Specifying the System Properties

At runtime, a set of temporal properties need to hold to ensure that the system works and adapts properly. To represent these properties, we adopt property specification patterns [56]. These patterns provide an easy-to-use notation for defining commonly occurring system properties that would otherwise require formal notations such as computational tree logic (CTL) [40] for their specifications. Example patterns are: (1) Absence that means an event does not occur during the system execution; (2) Precedence that means an event should always be preceded by another event during the system execution. Each pattern also has a scope to define when the pattern should hold. For example, the scope “Globally” specifies that a pattern must hold during the entire system execution, while the scope “After” means a pattern should hold after the first occurrence of an event. More details about these patterns can be found in [56]. We do not adopt the Object Constraint Language (OCL) to capture the system’s temporal properties because it is only able to specify static properties [172-173]. In addition, a number of extensions to the OCL have been introduced to capture temporal properties (e.g., [174-175]). However, they cannot be easily used to specify a system’s temporal properties, where these properties are specified at a low level of abstraction that is far from the natural language [104, 176].
Similar to capturing the system’s functional and adaptation requirements as scenarios, we extend the UML sequence diagram to allow the specification of the system properties in a form similar to the scenarios (following the property specification patterns [56]). We first introduce two new interaction operators: \textit{pattern} and \textit{pattern scope} to represent the property patterns and their scopes respectively (see Figure 4-11). Then, the \textit{property} operator is introduced to define combined fragments consisting of a pattern fragment (defined using the pattern operator) and optionally a pattern scope fragment (defined using the pattern scope operator). When the pattern scope is not specified, the default globally scope is assigned automatically to the property. The graphical representations for these operators and fragments are given in the right of Figure 4-12.

![UML Sequence Diagram – Core Elements](image)

**Figure 4-11:** A meta-model for the extended UML sequence diagram to capture the system properties

The system has multiple functional variants and it switches between them in response to context changes. Thus, the system has two types of properties: \textit{local} and \textit{global} [27]. The local properties need to hold only with a specific system variant, while the global properties need to be preserved in all system variants (i.e. independent of the context situations). To specify the context dependent (local) properties, we incorporate into the property specification patterns “\textit{context conditions}” that specify the context situations where these properties should hold as shown at the bottom of Figure 4-11.

Figure 4-12 shows some example system properties. First, the property “PO1” is a global property and should hold in all system variants, i.e., the logout operation should be always
preceded by the login operation. Second, the property “PO2” is a local property and should only hold when the route planner service is selected, meaning that the “ProvideRoutes2” interaction should exist (i.e. the pattern “ET1”) after the interaction “PlanRoute” is requested (i.e. the scope “AF1”). Third, the property “PO3” is a local property and specifies that the sequence reference “Find Attractions” must not exist when the user does not want the attractions finder.

![Diagram](image)

**Figure 4-12:** Example properties that need to be preserved while the travel guide system in operation

### 4.2 Validating the System Requirements

While a context-aware adaptive system in operation, it switches from one variant to another in response to context changes. During this adaptation, a set of system properties need to hold. As such, the system variants need to be enumerated and validated against the relevant properties.

To validate the system’s variant specifications, we automatically enumerate and generate the system variants from the scenarios. Then, we transform the generated variants to formal models to check their conformance to the relevant system properties as presented in Figure 4-13 and described in the next sections.
4.2.1 Enumerating and Generating the System Variants

To enumerate and generate the variant specifications of a context-aware adaptive system, we compute adaptation actions (as scripts) that need to be applied into the system functionality in response to context changes. Then, the scripts are checked to ensure their consistency. Finally, the consistent scripts are applied to the functional scenarios to generate the system variants as shown in Figure 4-13.

(1) Computing Adaptation Scripts: In response to context changes, the system switches from one variant to another by executing a set of adaptation actions (as a script). Thus, to generate the system variants, different adaptation scripts in response to context changes need to be identified. To compute the adaptation scripts, we first enumerate the possible context situations (or context conditions) that trigger the system’s runtime adaptation. We identify these conditions by parsing
the adaptation scenarios (see Figure 4-13). For each condition specified in the scenarios, its existence in the accumulating list of conditions is checked. If the condition is new (i.e. this is the first appearance of the condition in the scenarios), it is added to the conditions list (Lines 2-9 of Listing 4-1). The result of this step is a set of conditions that trigger the system adaptation. For example, applying this part of the algorithm to the adaptation scenario “AS1” shown in Figure 4-8 generates the following list with two conditions:

```
[(FindingAttractionSelected==True),
(RoutePlannerSelected==True)]
```

Second, combinations of changes to the context variables that trigger the system adaptation are unpredictable at the design time. As such, after collecting the conditions that can trigger the system adaptation, we generate all combinations of these conditions and remove the invalid ones. Each condition can be true or false, and then the number of the combinations equals to $2^n$ where $n$ is the number of the identified context conditions. To assign Boolean values to one condition combination, we use the binary representation of the combination index with $n$ binary digits (see Lines 10-14 in Listing 4-1). For example, the second combination “Combination2” of the conditions generated from the scenario “AS1” is “01” (i.e. the index is 1) which means:

```
[“FindingAttractionSelected==True” is false,
 “RoutePlannerSelected==True” is true].
```

Then, we identify and remove the invalid combinations (i.e. combinations that have conflicting conditions that cannot hold specific Boolean values at the same time) as shown in Line 15 of Listing 4-1. For example, the traffic information is either available or not. Thus, the two context conditions “TrafficInfoAvailability==True” and “TrafficInfoAvailability==False” specified in Figure 4-9 cannot be true or false at the same time, and then these invalid combinations are identified and removed.

Third, the valid combinations of adaptation conditions are used to generate the adaptation scripts (each of which corresponds to a system variant). For each combination, we traverse the adaptation fragments in the adaptation scenarios to identify executable ones (i.e. their conditions are evaluated to true). Then, the adaptation actions specified in the executable fragments are added to an adaptation script as shown in Lines 16-23 of Listing 4-1. For example, using the condition combination “Combination2” described above, the condition of the fragment “AD1” in the scenario “AS1” (shown in Figure 4-8) is evaluated to false while the condition of the fragment “AD2” is evaluated to true. Thus, the following two actions as a script (i.e. “S2”) need be applied into the scenario “FS1” when the combination “Combination2” holds:
“[RemoveSequenceReference (FS1, RF1),
AddSequenceReference (FS1, RF2)].

Listing 4-1: An algorithm (pseudocode) to generate and validate the system variants

```csharp
1: Void ValidateRequirements (FunctionalScenarios FS, AdaptationScenarios AS)
   // Compute the adaptation conditions
2:   List ConditionsList = new List();
3:   FOR each Scenario S in AS
4:     FOR each AdaptationCondition AC in S
5:       IF AC does not exist in the ConditionsList THEN
6:         ConditionsList.add(AC);
7:     END IF
8:   END FOR
9: END FOR
   // Generate the possible condition combinations
10: List ConditionsCombinations = new List();
11: ConditionsCombinations.setSize(power(2, sizeOf(ConditionsList));
12:   // E.g. three adaptation conditions have eight combinations
13:   FOR each Combination C in ConditionsCombinations
14:     C.assignBooleanValue(BooleanValueOf(C.index()));
15:     // The assignment is done through the binary representation of the
16:     // combination index. E.g. if we have three conditions [C1, C2, C3], then
17:     // the second combination (i.e. index is 1, where the index start with 0)
18:     // will be [False, False, True] which corresponds to [001]
19:   END FOR
20:   ConditionsCombinations.RemoveInvalidCombinations();
21:   // Remove combinations with conflicting conditions having the same value.
22:     // E.g. if C1 is A>1 and C2 is A<1,
23:     // then the combination [C1=True, C2=True] is invalid.
24:   // Generate and validate adaptation scripts
25:   List AdaptationScripts = new List();
26:   FOR each Combination C in ConditionsCombinations
27:     AdaptationScript script = new AdaptationScript();
28:     List ExecutableFragments = new List();
29:     ExecutableFragments = AS.getExecutableAdaptations();
30:     // This is done by evaluating the adaptation fragments’ conditions based
31:     // on Boolean values of the conditions in the combination. A fragment is
32:     // executable when its condition is evaluated to true.
33:     FOR each Fragment EF in ExecutableFragments
34:       script.addAction(EF.getAction());
35:     END FOR
36:     // Adding scripts that have consistent adaptation actions. For example, a
37:     // script that has conflicting actions such as AddMessage “FM1” and
38:     // RemoveMessage “FM1” (i.e. add and remove the same message) is inconsistent.
39:     IF script.isConsistent() THEN
40:       AdaptationScripts.add(script);
41:     END IF
42:   END FOR
43:   // Generate and validate the system variants
44:   List SystemVariants = new List();
45:   FOR each Script ST in AdaptationScripts
46:     ST.orderAdaptationActions();
47:     // Ordering of the script is required to ensure its applicability. E.g. A life
48:     // line can be only removed after removing messages that it is involved in.
49:     Variant V = applyScript(FS, ST);
50:     V.addPropertiesToBeChecked();
51:     V.validate();
52:   END FOR
53: END
```
(2) Checking the Adaptation Scripts: An adaptation script is consistent, if it is free from redundancy, conflict, and incompleteness in its adaptation actions. First, in response to a context change, an adaptation action may be fired by two adaptation fragments. The application of this action to a system scenario twice leads to an invalid scenario (when an element is added twice to the scenario), or to a failure in the process of applying the script to the system scenarios when a removed element need to be removed again. Thus, the redundant actions need to be detected and removed.

Second, a generated script may have two actions to add and remove the same element or to modify an element twice in a scenario. These adaptation actions should not be fired at the same time, where applying one of the two actions to the system scenarios contradicts the application of the other (i.e. conflicting actions). For example, due to errors in specifying the adaptation scenarios, two actions to remove and add the sequence reference “RF1” (shown in Figure 4-4) are fired at the same time in response to a combination of changes to the context variables. As such, each generated script needs to be parsed to detect conflicts in its adaptation actions.

Third, in an adaptation script, a set of actions that must appear in the script may not be fired (i.e. the script is incomplete). These missing actions can be inferred from dependencies between the script actions. In general, a combined fragment depends on its messages, and a message depends on the lifelines involved. As such, to remove an element from a scenario, elements that depend on this element should be removed first. For example, to remove the last message in the alternative fragment “AT1” presented in Figure 4-6, this fragment should be removed first. Similarly, to add an element to a scenario, elements that this element depends on should be added first. For example, before adding the message “CM1” shown in Figure 4-6, the lifelines that send and receive this message (i.e. the route planner and the traffic information participants) must be added first. These action dependencies can be used for detecting missing actions, where dependent actions should coexist with each other. Thus, each generated script needs to be parsed to identify actions whose dependent actions do not exist in the script.

To ensure the consistency of the generated adaptation scripts, we parse the scripts to identify redundancy, conflict, and incompleteness in their actions (Line 24 in Listing 4-1). Then, for each inconsistent script, the software engineer is notified by the adaptation scenarios that lead to this inconsistent script, so that he can perform the required changes to the system’s adaptation scenarios and re-check them (see Figure 4-13). The adaptation scenarios of the travel guide system (described above) are specified carefully, and then the scripts generated from them are consistent. To show the ability of our algorithm in identifying inconsistencies in the adaptation scenarios, we have added an adaptation fragment to the scenario “AS1” shown in Figure 4-8.
The condition of this fragment is same as the condition of the fragment “AD2” in the scenario “AS1”, but its adaptation action is “RemoveSequenceReference (FS1, RF2)” that contradicts with the action of “AD2”. Using our algorithm, different adaptation scripts are enumerated and generated. However, there is a contradiction between the actions of two adaptation fragments in “AS1”, and then half of the generated scripts are inconsistent where the two fragments cannot be executed at the same time and this error is reported to the software engineer.

Figure 4-14: The different variants of the scenario “FS1” generated by the adaptation scenario “AS1”

(3) Deriving the System Variant Specifications: To generate the system variants from the scenarios, the consistent scripts from the above step are applied to the system’s functional scenarios. For each consistent adaptation script, its adaptation actions are first automatically ordered to take into account the dependencies between the actions. Then, a system variant is generated by applying the script to the functional scenarios (Lines 28-31 in Listing 4-1). For example, the adaptation scenario “AS1” (shown in Figure 4-8) has two context conditions (RoutePlanner selected or not and FindingAttraction selected or not), and then there are 4 \(2^2\) condition combinations of them which leads to four scripts that can be derived from this adaptation scenario as shown in Figure 4-14. These four scripts are consistent, and then four variants of the functional scenario “FS1” (shown in Figure 4-4) can be generated by applying the four scripts to this scenario.

Example two variants of the functional scenario “FS1” are shown in Figure 4-15. The variant “Variant2” shown in Figure 4-15.A is the result of applying the script “S2” (shown in Figure 4-14) to the scenario “FS1”, where the sequence references “RF1” (find attractions) is removed in response to the user request to remove it. However, the action “AddSequenceReference RF2” does not has effect on the scenario “FS1”, because the designed scenario already contains all elements that the system needs to have at runtime as shown in Figure 4-4. Similarly, the variant “Variant3” shown in Figure 4-15.B is generated by applying the script “S3” (presented in Figure 4-14) to the scenario “FS1”.

84
While the system is in operation, it needs to switch from one variant to another to cope with context changes. The adaptation scenarios are only able to generate the system variants that suit different context situations from the functional scenarios (as discussed above). Thus, to switch between two system variants in response to a context change, a number of steps are performed in our approach. First, the adaptation scenarios are used for generating an adaptation script to cope with the context change. Second, the generated script is applied to the functional scenarios to generate a system variant (i.e. the target variant). Finally, differences between the current and target variants are computed to specify what actions need to be applied into the current variant. For example, to switch from the variant “Variant2” to the variant “Variant3” of the functional scenario “FS1”, the following two actions need to be carried out:

```
[AddSequenceReference (FS1, RF1), RemoveSequenceReference (FS1, RF2)]
```

These two actions represent the differences between the two variants as shown in Figure 4-16. Figure 4-16 shows the four variants that are generated from the scenario “FS1”, and possible adaptation actions to switch between these variants at runtime. In general, the system is able to switch from any variant to another, where all system variants are generated from the same set of functional scenarios (i.e. they have the same base model).
The algorithm shown in Listing 4-1 generates the system variants from multiple scenarios. But, in the above, we only gave an example of how to use an adaptation scenario to generate the variants of a functional scenario. Thus, we discuss below the application of our algorithm to the functional and adaptation scenarios of the travel guide system to generate its full variants.

The travel guide system is captured as three functional scenarios (see Figure 4-4, Figure 4-5, and Figure 4-6) and three adaptation scenarios. Two of the adaptation scenarios are presented in Figure 4-8 and Figure 4-9. The third adaptation scenario is about changing the scenario “FS2” in response to changes in the weather information availability. In this scenario, the message “CM1”, the lifeline “CL1”, and the alternative fragment “AT1” are removed when the traffic information is not available while they are added otherwise. Using these adaptation scenarios, 5 context conditions are identified, and then there are 32 condition combinations. The two context conditions shown in Figure 4-9 have a conflict, and then half of these combinations are removed leading to only 16 valid combinations. The adaptation scenarios are specified carefully with no inconsistency in their adaptation actions, and then 16 consistent scripts with their corresponding 16 variants are generated. Example adaptation scripts to be applied to the functional scenarios of the travel guide system to generate some of its variants are shown in Figure 4-17.
At runtime, the travel guide system needs to switch between the 16 variants to cope with the context changes. As discussed above, the switching between two system variants in response to a context change is performed by identifying a target variant that suit the context change. Then, adaptation actions corresponding to the differences between the current and target variants are generated and applied to the current variant, so that the system moves to the target variant. An example switching between three variants of the travel guide system is shown in Figure 4-18.

4.2.2 Validating the System Variant Specifications

To validate the system variants that are generated from the scenarios, we identify the properties that need to be checked against these variants. Then, the system variants and their properties are transformed to formal models to enable their validation.

(1) Identifying the Relevant Properties: To identify system properties that need to be checked against a generated variant, we first parse the system properties (see Line 32 in Listing 4-1). If a property does not have a context condition (i.e. global property), then it should be satisfied by this variant (e.g. the precedence property “PD1” shown in Figure 4-12). On the other hand, if a property needs to hold only in a specific context situation (i.e. local property), then the context situation in which this variant needs to be active is compared with the property’s context condition. If the property condition is satisfied, then this property needs to hold in this system variant (e.g. the property “AB1” in Figure 4-12). Example properties that need to be checked against the variants of the functional scenarios “FS1” are shown at the bottom of Figure 4-15.

(2) Validating the Variants Formally: In order to validate the system variants against the relevant system properties, we transform the variants and their properties to formal models (see the bottom part of Figure 4-13). In this regard, we choose to use the Romeo tool to perform the
formal validation [41]. Therefore, we transform the system variants to Petri nets and the system properties to computational tree logic (CTL). First, we adopt the approach proposed by Bernardi et al. [177-178] to transform each scenario in a system variant to a Petri net. For example, the variants “Varaint2” and “Varaint3” of the functional scenario “FS1” (shown in Figure 4-15) is transformed to their corresponding Petri nets as shown in Figure 4-19 (more details about this transformation process can be found in [177]).

Second, we use the mappings proposed by Dwyer et al. to transform the properties to CTL formulas [56]. For example, the precedence property “PD1” between login and logout shown in Figure 4-12 is transformed to “\( \text{not E [not login U (logout and not login)]} \)”. Finally, the Petri nets and the CTL formulas are fed into the Romeo tool for validation (Line 33 of Listing 4-1). If the properties are not satisfied by the system variants, the software engineer is alerted, and corrective actions need to be carried out to the adaptation and/or the functional scenarios as shown in the bottom of Figure 4-13.

Following the above steps, the variant specifications of the travel guide system are generated and validated against the relevant properties. The system scenarios are simple and we specified them carefully. Therefore, the validation results show that all system properties are satisfied by its variant specifications (i.e. all system variants are valid). To demonstrate an example property
violation, we injected an error into the adaptation scenario “AS1” where the sequence reference “RF1” (find attractions) in the scenario “FS1” is not removed when the attractions finder service is not needed by the user. As such, when the property “AB1” (shown in Figure 4-12) is checked against the variant “Variant2” of the scenario “FS1” that is generated by the modified adaptation scenario, the validation result shows that the property “AB1” is violated by this variant and then the adaptation scenario “AS1” need to be corrected (modified) to resolve this error.

4.3 Summary

In this chapter, we have introduced a scenario-based approach to assist the software engineer in specifying and validating the requirements of a context-aware adaptive software system. First, our approach specifies the system’s functionality as a set of functional scenarios. In particular, we identify environmental context and its functional use in these scenarios. In addition, we have proposed the concept of adaptation scenarios to capture the requirements for the system runtime adaptation. These scenarios specify adaptations to the system’s functional scenarios in response to context changes. We also represent the system properties that need to hold when the system adapts at runtime in a form similar to the scenarios. To specify the two types of scenarios and the system properties, we have extended the UML sequence diagram.

Second, the system’s functional variants are automatically enumerated and validated against the expected system properties for conformance. To do so, the adaptation scenarios are used to generate a set of adaptation scripts that need to be applied to the system in response to context changes. Then, these scripts are checked to ensure the consistency of the adaptation scenarios, and the consistent scripts are applied to the functional scenarios to generate possible variant specifications of the system. Finally, the system variants and properties are transformed to Petri nets and computational tree logic, before being fed into an existing model checker for the conformance check.

The contribution of this chapter is twofold: (1) a scenario-based approach to specifying the requirements and properties of context-aware adaptive systems; (2) techniques to automatically check the consistency of a system’s adaptation scenarios, and validate that the application of the adaptation scenarios to the functional scenarios does not lead to invalid system variants.
Chapter 5: Design and Realization of Context-aware Adaptive Software Systems

After capturing the requirements of a software system, a design model for the system needs to be specified [179-180]. This model contains a set of elements that interact with each other to meet the system requirements [181-182]. In addition, to maintain a casual connection between the system requirements and its design and to reduce the effort required for designing the system, the design model of the system needs to be synthesized from the requirements [19]. Furthermore, to realize the software system, the design model needs to be transformed to an executable model for the deployment [44, 183].

A context-aware adaptive system is required to have a high variability because it operates in a dynamic environment [6]. In addition, the system consists of three aspects (i.e. functionality, context, and management) that are highly connected to each other [1]. As discussed in Section 3.4, in existing approaches (e.g. [16, 27-28]), the system’s three aspects and their relationships are not clearly identified and consequently designed and realized. As such, these aspects and their relationships cannot be clearly captured, or easily changed in response to anticipated and unanticipated changes. In addition, to design a context-aware adaptive software system by the existing approaches, the software engineer is responsible for translating the system requirements to a design model manually. Thus, the causal connection between the system’s design model and its requirements is maintained manually, which is a complex task when the system is large. To tackle these challenges, in this chapter, we present a novel method to design a context-aware adaptive system. This method explicitly models the system from its three aspects, so that the aspects and their relationships can be clearly captured. To further ease the task of designing the system, we introduce a technique to synthesize the system’s design model from its requirements (specified as scenarios as discussed in Chapter 4). We also introduce a technique to transform the design model to an executable model for the deployment, and adopt models@runtime [135] to enable the system adaptation in response to anticipated and unanticipated changes.

This chapter is organized as follows. An overview of our process to designing and realizing context-aware adaptive systems is discussed in Section 5.1. Section 5.2 presents a method to design context-aware adaptive systems. In Section 5.3, we introduce a technique to synthesize the system’s design model from its scenarios description. Based on the synthesized model, the software engineer can add elements that are related to the system’s solution space but cannot be synthesized from the system scenarios. Section 5.4 describes a technique for transforming the system’s design model to an executable model for the deployment.
5.1 Designing and Realizing Context-aware Adaptive Systems

A two steps process for designing and realizing a context-aware adaptive software system based on its requirements is shown in Figure 5-1. The first step concerns the translation of the system requirements to a design model. The second step is about transforming the system’s design model to an executable model for the deployment. To support the software engineer in doing the two steps, we introduce a method to design the different aspects of the system, and techniques to synthesize the system design model from its requirements and to transform the design model to an executable model for the deployment.

![Figure 5-1: A process for designing and realizing context-aware adaptive systems](image)

*Designing Context-aware Adaptive Systems:* In designing a context-aware adaptive system, three system aspects (i.e. functionality, context, and management) need to be explicitly captured to enable the system adaptation. Following an *organizational* approach in designing a software system, the system is modelled as a number of *roles* that interact with each other by *contracts* to capture the roles’ relationships explicitly so that the system can be clearly designed and easily manipulated (adapted) at runtime in response to context changes [45-46]. As such, we adopt an organizational approach to design a context-ware adaptive system as two composites: functional and management. The *functional* composite captures the system functionality. The *management* composite specifies adaptations to the functional composite in response to context changes.

First, the functional composite is designed as set of *functional roles* that interact with each other through *functional contracts* to capture the system functionality as shown in Figure 5-2. In

---

3 A role is an abstract definition of tasks that need to be performed by the role while there are players who actually perform these tasks by playing that role.
addition, to make the system’s functionality context-aware, there is a set of context roles bound to context providers to make the context information available. The functional composite also has contextual contracts that are formed between context and functional roles to specify context information required by the functional roles to continue their operations. Furthermore, to define the system’s functional behaviour, a set of behaviour processes are specified. In designing these processes, we adopt an event-based approach [184-185] where a process is designed as loosely coupled tasks that are related with each other through events, so that the process can be easily adapted at runtime. An example event-based process is “Process1” shown in Figure 5-2.

**Figure 5-2:** Example functional and management composites of a context-aware adaptive system

Second, based on the organizational approach, the management composite has three types of roles and two types of contracts as shown in Figure 5-2. The three types of roles are: functional system, context, and management. The functional system role represents the system’s functional composite (see Figure 5-2) that needs to be adapted at runtime. The context roles provide context information that triggers the system adaptation. The management role is responsible for deciding a set of adaptation actions in response to context changes. The two types of contracts are: management and contextual. The management contract is formed between management and functional system roles to capture adaptation actions that need to be applied on the system. The contextual contracts define which context information that triggers the system adaptation.
Synthesis of the System Design from its Requirements: To ease the task of the system design and maintain a casual connection between the system design and its requirements, we introduce a technique to synthesize the system’s design model from its requirements that are specified as a set of scenarios (see Chapter 4). In general, the functional scenarios are mapped to the system’s functional composite, and the management composite is derived from the adaptation scenarios as shown in Figure 5-1. First, to synthesize the functional composite, functional and contextual participants in the functional scenarios are mapped to functional and context roles respectively. Interactions (messages) between the scenarios’ participants are used for deriving functional and contextual contracts between the synthesized roles. The scenarios flows are transformed to behaviour processes that represent the system’s functional behaviour.

Second, roles of the management composite are derived from participants in the adaptation scenarios. For example, a management participant in the adaptation scenarios is translated to a management role in the system design. In addition, contextual and management contracts of this composite are derived from messages that are exchanged between the adaptation scenarios’ participants. Furthermore, adaptation fragments in the adaptation scenarios are translated to a set of adaptation rules to decide the system’s runtime adaptation in response to context changes.

Realizing the System: To realize context-aware adaptive systems designed in our approach, we adopt the ROAD framework which is an extension to Apache Axis2\(^4\) to realize adaptive software systems [47]. To realize a context-aware adaptive system using this framework, we transform the system’s design model to an executable model that is compatible with the ROAD framework (see Figure 5-1). When the executable model is deployed to the ROAD runtime environment, an instance of the system is created. This instance maintains a runtime model for the system using the models@runtime concept to support the application of different adaptation actions to the system functionality. For example, the functional composite of the system is able to add, remove, or modify its roles and contracts.

5.2 Designing Context-aware Adaptive Software Systems

A context-aware adaptive system has three aspects: functionality, context, and management. The system functionality represents tasks (functions) that need to be provided by the system. The system management is responsible for adapting the system in response to context changes. The context aspect specifies context information that is needed by the system functionality or that triggers its runtime adaptation. Therefore, to design such system, the three aspects need to be explicitly modelled, so that they can be clearly captured and easily manipulated.

\(^4\) [http://axis.apache.org/axis2/java/core/](http://axis.apache.org/axis2/java/core/)
A software system as an organisation is a set of dynamic relationships between its roles to maintain the system viability in a changing environment [45-46, 186]. In such a view, first, the relationships (defined as contracts) specify the possible interactions between the system roles. Second, the roles are abstract definitions of tasks that should be provided by the system. The descriptions of these roles are derived from the contracts, where the interactions defined in the contracts are used for specifying what tasks the system roles should do. Third, the system has a set of players who actually perform the roles’ tasks by playing the roles. Fourth, to coordinate interactions between the system roles to achieve composite tasks, a set of behaviour processes are specified. The processes define in what order a set of simple tasks are performed to achieve the composite tasks. Fifth, the system has a manager (an organizer) role. This role is responsible for changing the system roles, bindings of the players to the roles, and the relationships between the roles to maintain the system viability in the face of environment changes.

The organizational approach represents the system’s different elements and the relationships between them explicitly so that the elements and their relationships can be clearly captured [65]. It also keeps the system structure alive at runtime, so that the system can be easily manipulated (adapted) to cope with context changes [130]. Therefore, we adopt the organizational approach to design context-aware adaptive software systems.

Following the organizational approach, we design a context-aware adaptive software system as two main composites: functional and management. The functional composite represents the system’s core functionality, while the management composite captures the runtime adaptation to the system functionality in response to context changes. In the following, we describe these two composites in detail.

5.2.1 The System’s Functional Composite

Based on the organizational approach, the functional composite (see Figure 5-3) consists of a set of functional roles that interact with each other through functional contracts (i.e. the functional structure). In addition, to make the system functionality context-aware, the composite contains a set of context roles bound to context providers to make the context information available (i.e. the context model). Furthermore, the composite has a set of behaviour processes to capture the system’s functional behaviour.

5.2.1.1 The Functional Structure

To capture the functional structure of a context-aware adaptive system, we model the structure as a set of functional roles that interact with each other by functional contracts (i.e. relationships
between the system roles). Thus, the system’s functional elements and their relationships can be clearly captured and easily manipulated at runtime. In addition, each role can be played by one or more functional players (see Figure 5-3), so that the role players’ binding can be changed at runtime in response to context changes.

**Figure 5-3:** A meta-model for a context-aware adaptive system’s functional composite

**Figure 5-4:** The travel guide system’s functional structure and context models

(A) *Functional Roles and their Players:* First, the functional roles represent the system’s functionality where each role position description is an abstract definition of tasks that this role should provide (i.e. the *role interface*). For example, the route planner role (shown in Figure 5-4
(B) **Functional Contracts**: The functional contracts capture interactions, functional system qualities (i.e. the non-functional requirements), and conversation clauses between the system’s functional roles (see Figure 5-3). Thus, each contract has the following items. First, the contract has an identifier and is formed between two functional roles (i.e. roles “A” and “B” as shown in Figure 5-3). For example, the contract “FC2” is formed between the user and the route planner roles as presented in Listing 5-1.

Second, the contract has a set of permissible *interactions* between the contracted roles as shown in Listing 5-2. Each interaction has (1) an *identifier* (e.g. “i1”); (2) an *operation* that need
to be executed by requesting that interaction and this operation has a name (e.g. “PlanRoute”) and input parameters (e.g. current location and destination); (3) a direction to specify who is responsible for providing the operation included in the interaction (e.g. “AtoB” which means the route planner is responsible for providing the route calculation); (4) a return type (e.g. void).

Third, to define the system’s non-functional requirements (e.g. response time and reliability), a set of functional system qualities are specified on the functional interactions. We follow the Web Service Level Agreement (WSLA) language in defining these qualities [187]. An example functional quality is “q1” (shown in Listing 5-2) that means “the routes calculation operation (task) should not take more than 7 seconds in calculating the routes when the traffic information is available”.

Listing 5-2: The functional contract “FC2”

<table>
<thead>
<tr>
<th>Functional Contract ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC2: User_RoutePlanner</td>
</tr>
</tbody>
</table>

Parties:
- Role A: User;
- Role B: RoutePlanner;

Interaction Clauses:
- $i_1$: {PlanRoute (String: CurrentLocation, String: Destination), AtoB, void};
- $i_2$: {ProvideRoutes1 (String: Routes), BtoA, void};
- $i_3$: {ProvideRoutes2 (String: Routes), BtoA, void};
- $i_4$: {SelectRoute (Integer: RouteId), AtoB, Boolean};

Functional Qualities:
- $q_1$: {i2, float: ResponseTime, LessThan , 7 seconds}
- $q_2$: {i3, float: ResponseTime, LessThan , 5 seconds}

Conversion Clauses (Temporal Constraints):
- $c_1$: {existence $i_3$ after $i_1$}
- $c_2$: {i1 precedes i3 globally}

Fourth, the contract has a set of conversation clauses that specifies the acceptable sequences of interactions between two functional roles. The Interaction Rule Specification (IRS) language is a language (that is based on temporal logic) to specify temporal constraints on the interactions between a system’s components [188]. It does not require that the software practitioners have a profound knowledge of formal methods (e.g. temporal logic [189]). Therefore, we have used the IRS language to specify the acceptable sequences of interactions between two functional roles. Each conversation clause captures the order of two functional interactions, and it has an order pattern such as “leads to” (i.e. must always be followed by) and “precedes” (i.e. must always be preceded by), and a scope such as “globally” (i.e. must hold during the entire system execution) and “after” (i.e. must hold after the first occurrence of another clause). More details about the IRS language can be found in [188]. An example clause is “c2” (shown in Listing 5-2) meaning the “PlanRoute” operation must be invoked before “ProvideRoutes2” operation during the entire system execution.
5.2.1.2 The Context Model

To capture the context information that is needed by the system functionality, a set of contextual contracts are specified in the functional composite (see Listing 5-1). These contracts are formed between context sources and context consumers. The context sources are context roles that are responsible for providing the context information, while the context consumers are functional roles that use this information. In addition, there are context providers bound to the context roles to make the context information available (see Figure 5-3). As such, the context roles and their relationships with the functional roles can be clearly captured and easily manipulated.

(A) Context Roles and Context Providers: The context roles specify context entities that the system needs to know information about. The descriptions of these roles are derived from their contextual contracts (discussed below), where these roles provide context attributes specified in such contracts. In addition, each context role is bound to a context provider that is responsible for providing some context information. For example, a traffic information role bound with its player (e.g. a road side unit) is responsible for providing the live traffic information to the route planner role as shown in Figure 5-4 and Listing 5-1.

(B) Contextual Contracts: The contextual contracts define context information needed by the system’s functional roles and the quality of this required context (e.g. accuracy, freshness, etc.). For example, the contract “CC1” shown in Listing 5-3 specifies that the route planner role needs to know the live traffic information to calculate the routes effectively. Similar to specifying the functional system qualities, we follow the WSLA language in defining the context qualities (e.g. the traffic information freshness needs to be less than two minutes as shown in Listing 5-3).

\[
\text{Listing 5-3: The contextual contract “CC1”}
\]

<table>
<thead>
<tr>
<th>Contextual Contract ID</th>
<th>CC1: TrafficInfo_RoutePlanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties:</td>
<td>Context Source: TrafficInfo;</td>
</tr>
<tr>
<td></td>
<td>Context Consumer: RoutePlanner;</td>
</tr>
<tr>
<td>Context Attributes:</td>
<td>(a1: \text{String}: \text{TrafficInformation};)</td>
</tr>
<tr>
<td></td>
<td>Context Attributes’ Qualities:</td>
</tr>
<tr>
<td></td>
<td>(q1: {a1, \text{float}: \text{Freshness}, \text{LessThan}, 2 \text{ minute}})</td>
</tr>
</tbody>
</table>

5.2.1.3 The System’s Functional Behaviour

The functional behaviour model captures how the system behaves to provide its functionality at runtime. In response to a user request, the system executes a simple or a composite task. First, the request of simple tasks is supported directly by the contracts (i.e. bidirectional interactions
between two roles), where the roles’ tasks are derived from the contracts. The interactions of the functional contracts specify tasks in the functional roles, where based on an interaction direction a task is added to a role. For example, the user can request the interactions “PlanRoute” and “SelectRoute” from the route planner role, and then this role has two tasks corresponding to the two interactions as presented in Listing 5-4. Similarly, the context attributes specified into the contextual contracts define tasks in the functional and the context roles. The context information can be requested by (or notified to) a functional role. Thus, for each context attribute two tasks are derived. The first task is added to the context source role, so that other roles can request the context information (e.g. the task “GetTrafficInfo” in Listing 5-4). The second task is added to the context consumer role, where the consumer can be notified by the context change (e.g. the task “NotifyTrafficInfo” in Listing 5-4).

Listing 5-4: The tasks of the route planner and the traffic information roles

<table>
<thead>
<tr>
<th>Role FR3: RoutePlanner</th>
<th>Tasks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1: PlanRoute {...};</td>
<td>// Plan a route from a location to destination.</td>
</tr>
<tr>
<td>t2: SelectRoute {...};</td>
<td>// Select a route for the journey.</td>
</tr>
<tr>
<td>t3: NotifyTrafficInfo {...};</td>
<td>// Notify the route planner by the changes in the traffic information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role CR3: TrafficInfo</th>
<th>Tasks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1: GetTrafficInfo {...};</td>
<td>// Acquiring the traffic information from its provider</td>
</tr>
</tbody>
</table>

Second, to support the request of composite tasks, a set of behaviour processes are defined in the system’s functional composite as shown in Figure 5-5. These processes define in what order a set of tasks are executed to provide the composite tasks. The processes also have constraints that need to hold at runtime to ensure that they are working properly.

Figure 5-5: A meta-model for a system’s behaviour process

(A) Behaviour Processes: In designing the behaviour processes, we follow an event-based approach [184]. We adopt the event-based approach because it represents a process as a set of
loosely coupled tasks that are related to each other through events, so that the process can be easily adapted at runtime by manipulating the pre and post events of its tasks. The process tasks can be simple or composite as shown in Figure 5-5. A simple task is corresponding to a single task (function) execution, while a composite task is a behaviour process itself.

Each task has pre (events that enable the execution of a task) and post (events that are generated upon a task completion) conditions as a set of events. The pre and post conditions can be simple (i.e. single event) or complex (i.e. a set of events that are combined with each other by logical operators such as “and”, “or”, and “xor” as shown in Figure 5-6). The process also has two events that specify the process’s start and end. An example process is “P3: PlanRoute” shown in Listing 5-5. To ease the understandability of an event-based process, we visualized the process in the form of an Event-driven Process Chain (EPC) [184] (see Figure 5-6). The process “PlanRoute” starts by a user request to plan a route from a location to destination. To suggest a route, the live traffic information and the user selected attractions are then acquired from their providers (i.e. the tasks “t2” and “t3” in Listing 5-5). Based on the user selected attractions, a suitable route planning function is selected where “ProvideRoutes1” is used when attractions are selected by the user and “ProvideRoutes2” is used otherwise. The two functions take the traffic information into account during the routes calculation. Finally, a number of routes are suggested to the user where he can select a suitable route and starts his journey.

Listing 5-5: The description of the “P3: PlanRoute” process

<table>
<thead>
<tr>
<th>Behaviour Process P3: PlanRoute</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start Event:</strong> PlanRouteRequested;</td>
</tr>
<tr>
<td><strong>End Event:</strong> RouteSelected;</td>
</tr>
<tr>
<td><strong>Tasks:</strong></td>
</tr>
<tr>
<td>t1: {FR3.PlanRoute, Pre (PlanRouteRequested), Post (CalculateRoutes)};</td>
</tr>
<tr>
<td>t2: {CR3.GetTrafficInfo, Pre (CalculateRoutes), Post (TrafficInfoAvailable)};</td>
</tr>
<tr>
<td>t3: {CR2.GetSelectedAttractions, Pre (TrafficInfoAvailable), Post (SelectedAttractionsProvided)};</td>
</tr>
<tr>
<td>t4: {FR1.ProvideRoutes1, Pre (SelectedAttractionsProvided and AttractionsSelected), Post (RoutesProvided)};</td>
</tr>
<tr>
<td>t5: {FR1.ProvideRoutes2, Pre (SelectedAttractionsProvided and AttractionsNotSelected), Post (RoutesProvided)};</td>
</tr>
<tr>
<td>t6: {FR3.SelectRoute, Pre (RoutesProvided), Post (RouteSelected)};</td>
</tr>
<tr>
<td><strong>Temporal Constraints:</strong></td>
</tr>
<tr>
<td>c1: {t2 precedes t4 globally}</td>
</tr>
<tr>
<td>c2: {t6 response to t5 globally}</td>
</tr>
</tbody>
</table>

(B) Process Constraints: While a process is executing, a set of system temporal properties (constraints) need to hold to ensure that the system works properly. To specify these constraints, we follow the IRS language [188]. For example, to ensure that the process “P3” (visualized in Figure 5-6) works properly, the constraints “c1” and “c2” need to be preserved at runtime. The constraint “c1” means that the “GetTrafficInfo” task must precede the “ProvideRoutes2” task during the execution of the “PlanRoute” process.
5.2.2 The System’s Management Composite

To capture the system adaptation in response to context changes, we introduce the system’s management composite. Based on the organizational approach, this composite consists of three types of roles and two types of contracts as shown in Figure 5-7. First, the three types of roles are: functional system, context, and management. The functional system role represents the functional composite that need to be adapted (e.g. the travel guide system shown in Figure 5-8). The context roles represent context information that causes the system’s runtime adaptation (e.g. the user information role shown in Listing 5-6). The management role bound with its player is responsible for deciding the required adaptation actions in response to context changes (e.g. the system’s organizer role and its player shown in Listing 5-6).

Second, to capture relationships between the composite roles, two types of contracts are used: management and contextual. A management contract is formed between a management role and a functional system role to specify adaptation actions that can be applied to the system (e.g. the contract “MC1” in Figure 5-8). A contextual contract is formed between a management role and a context role (e.g. the contract “CC1” in Listing 5-6) to capture context information.
that triggers the system adaptation. We discuss below the functional system and management roles, the organizer player, and the management contract. The context roles and the contextual contracts are same as those discussed above where they are common between the functional and the management composites, and then we do not discuss them.

**Figure 5-7**: A meta-model for a context-aware adaptive system’s management composite

**Figure 5-8**: The travel guide system’s management composite

(A) **Functional System Role**: The functional system role represents the system functionality (captured by the functional composite) that needs to be adapted in response to context changes. For example, the functional composite of the travel guide system (shown in Figure 5-4) plays the “SystemFunctionality” role as presented in Listing 5-6.
Listing 5-6: A high level description of the management composite for the travel guide system

Management Composite \textit{TravelGuideSystemManagement}

Management Role:
\begin{itemize}
  \item MR1: SystemOrganizer \{\ldots\};
\end{itemize}

Functional System Role:
\begin{itemize}
  \item SR1: SystemFunctionality \{\ldots\};
\end{itemize}

Context Roles:
\begin{itemize}
  \item CR1: UserInfo \{\ldots\};
  \item CR2: ContextInfoAvailability \{\ldots\};
\end{itemize}

Contextual Contracts:
\begin{itemize}
  \item CC1: UserInfo\_SystemOrganizer \{\ldots\};
  \item CC2: ContextInfoAvailability\_SystemOrganizer \{\ldots\};
\end{itemize}

Management Contract:
\begin{itemize}
  \item MC1: SystemOrganizer\_SystemFunctionality \{\ldots\};
\end{itemize}

Players:
\begin{itemize}
  \item OP1: OrganizerPlayer \textbf{CanPlay} SystemOrganizer;
  \item SC1: TravelGuideSystemFunctionalComposite \textbf{CanPlay} SystemFunctionality;
\end{itemize}

Context Providers:
\begin{itemize}
  \item CP1: TravelGuideDB \textbf{CanProvide} UserInfo;
  \item CP2: ContextProvidersSensor \textbf{CanProvide} ContextInfoAvailability;
\end{itemize}

(B) Management Role and its Player: The organizer (management) role is responsible for adapting the functional system in response to context changes to keep meeting the users’ needs. To decide adaptation actions in response to context changes, we model the organizer role player as Event-Condition-Action (ECA) rules [190]. We adopt the rule-based approach to capture the system reactions to context changes because of its expressiveness and availability of the tool support. A rule’s event is usually a context change that causes the system’s runtime adaptation. The rule condition specifies a context situation that needs a system reaction(s). The rule action is adaptation actions that need to be applied to the functional system (captured by the functional system role) in response to the context change. In general, the adaptation actions are to \textit{add}, \textit{remove}, or \textit{change} a system element. The following are possible actions that can be performed on the system’s functional composite:

\textit{Adaptation Actions on the Composite Roles:}
\begin{itemize}
  \item AddRole (RoleId, RoleName)
  \item RemoveRole (RoleId)
  \item Bind (RoleId, PlayerId)
  \item Unbind (RoleId, PlayerId)
\end{itemize}

\textit{Adaptation Actions on the Functional Contracts:}
\begin{itemize}
  \item AddFunctionalContract (ContractId, RoleA, RoleB)
  \item RemoveFunctionalContract (ContractId)
  \item AddInteractionToContract (ContractId, FunctionalInteraction)
  \item RemoveInteractionFromContract (ContractId, FunctionalInteractionId)
\end{itemize}
• ChangeInteractionParameters \( (\text{ContractId}, \text{FunctionalInteractionId}, \text{NewParameters}) \)
• ChangeInteractionName \( (\text{ContractId}, \text{FunctionalInteractionId}, \text{NewName}) \)
• ChangeInteractionReturn \( (\text{ContractId}, \text{FunctionalInteractionId}, \text{NewReturn}) \)
• AddFunctionalQuality \( (\text{ContractId}, \text{FunctionalQuality}) \)
• ChangeFunctionalQuality \( (\text{ContractId}, \text{FunctionalQualityId}, \text{NewFunctionalQuality}) \)
• RemoveFunctionalQuality \( (\text{ContractId}, \text{FunctionalQualityId}) \)
• AddConversationClause \( (\text{ContractId}, \text{ConversationClause}) \)
• ChangeConversationClause \( (\text{ContractId}, \text{ClauseId}, \text{NewConversationClause}) \)
• RemoveConversationClause \( (\text{ContractId}, \text{ConversationClauseId}) \)

Adaptation Actions on the Contextual Contracts:
• AddContextualContract \( (\text{ContractId}, \text{Source}, \text{Consumer}) \)
• RemoveContextualContract \( (\text{ContractId}) \)
• AddContextAttributeToContract \( (\text{ContractId}, \text{ContextAttribute}) \)
• RemoveContextAttributeFromContract \( (\text{ContractId}, \text{ContextAttributeId}) \)
• AddContextQuality \( (\text{ContractId}, \text{ContextQuality}) \)
• ChangeContextQuality \( (\text{ContractId}, \text{ContextQualityId}, \text{NewContextQuality}) \)
• RemoveContextQuality \( (\text{ContractId}, \text{ContextQualityId}) \)

Adaptation Actions on the Behaviour Processes:
• AddBehaviourProcess \( (\text{BehaviourProcessId}) \)
• ChangeProcessStartEvent \( (\text{BehaviourProcessId}, \text{Event}) \)
• ChangeProcessEndEvent \( (\text{BehaviourProcessId}, \text{Event}) \)
• AddEventToProcess \( (\text{BehaviourProcessId}, \text{Event}) \)
• RemoveEventFromProcess \( (\text{BehaviourProcessId}, \text{EventId}) \)
• AddTaskToProcess \( (\text{BehaviourProcessId}, \text{Task}) \)
• RemoveTaskFromProcess \( (\text{BehaviourProcessId}, \text{TaskId}) \)
• ChangeTask \( (\text{BehaviourProcessId}, \text{TaskId}, \text{NewTask}) \)
• ChangeTaskPreEvent \( (\text{BehaviourProcessId}, \text{TaskId}, \text{NewPreEvent}) \)
• ChangeTaskPostEvent \( (\text{BehaviourProcessId}, \text{TaskId}, \text{NewPostEvent}) \)
• AddTemporalConstraint \( (\text{BehaviourProcessId}, \text{NewConstraint}) \)
• RemoveTemporalConstraint \( (\text{BehaviourProcessId}, \text{ConstraintId}) \)
• ChangeTemporalConstraint \( (\text{BehaviourProcessId}, \text{ConstraintId}, \text{NewConstraint}) \)

An example adaptation rule is “TrafficInfoNotAvailable” shown in Listing 5-7. This rule is activated when the traffic information availability is changed (i.e. event). When this information
is not available (i.e. condition), the system adapts itself (i.e. actions) by removing the contextual contract “CC2”, binding the route planner role with the player “RoutePlanner2”, etc.

**Listing 5-7**: Parts of the description for the travel guide system’s organizer player

<table>
<thead>
<tr>
<th>Adaptation Rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule TrafficInfoNotAvailable</strong>:</td>
</tr>
<tr>
<td><strong>When</strong> ValueChanges (TrafficInfoAvailability);</td>
</tr>
<tr>
<td><strong>if</strong> TrafficInfoAvailability == False;</td>
</tr>
<tr>
<td><strong>Rule TrafficInfoAvailable</strong>:</td>
</tr>
<tr>
<td><strong>When</strong> ValueChanges (TrafficInfoAvailability);</td>
</tr>
<tr>
<td><strong>if</strong> TrafficInfoAvailability == True;</td>
</tr>
</tbody>
</table>

(C) **Management Contracts**: To apply the actions defined in the adaptation rules, the system functional composite should support the application of these actions. As such, a management contract between the organizer and the functional system roles is formed (see Figure 5-7). This management contract has a set of elements as follows. First, the contract contains management (adaptation) actions to be performed on the system at runtime. These actions are corresponding to the actions specified in the adaptation rules. An example management contract in the travel guide system is “MC1” shown in Listing 5-8. It has a set of the actions that need to be applied on the travel guide system such as the removal of the contact “CC2” (i.e. ma1), and the addition of the task “GetTrafficInfo” to the process “P3” (i.e. ma3). To specify the management actions, we follow the definition of the functional interactions. Each action has a type and a parameter to specify which element to be manipulated (e.g. remove the “TrafficInfo” role), a direction (e.g. “AtoB” which means the action is performed on the functional composite of the travel guide system), and a return (e.g. Boolean) as shown in Listing 5-8.

**Listing 5-8**: Part of the management contract “MC1”

<table>
<thead>
<tr>
<th>Management Contract ID:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MC1</strong>: SystemOrganizer_SystemFunctionality</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parties:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Role A</strong>: SystemOrganizer;</td>
</tr>
<tr>
<td><strong>Role B</strong>: SystemFunctionality;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Management Actions:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ma1</strong>: {RemoveContextualContract (“CC2”), AtoB, Boolean};</td>
</tr>
<tr>
<td><strong>ma2</strong>: {RemoveRole (“TrafficInfo”), AtoB, Boolean};</td>
</tr>
<tr>
<td><strong>ma3</strong>: {AddTaskToProcess (“P3”, “GetTrafficInfo”), AtoB, Boolean};</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State Transfer:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>st1</strong>: {ma3, (P3.getInstances()).ActiveEvents}</td>
</tr>
</tbody>
</table>
Second, to ensure safe system change, the management contract specifies states that should be transferred when the system adapts from a configuration (behaviour) to another. An example state that should be transferred is “st1” shown in Listing 5-8. It means that events that are active in any instance of the process “P3” before a system’s adaptation need to be transferred when the process “P3” is adapted by the adaptation action “ma3”. Thus, all instances of the process “P3” can continue their execution after the system adaptation properly.

A meta-model for a context-aware adaptive software system is shown in Figure 5-9. This meta-model is a combination of the meta-models for the system’s functional and management composites presented in Figure 5-3 and Figure 5-7 and described above.

Figure 5-9: A meta-model for a context-aware adaptive software system

---

5 To simplify the Figure, we have removed associations between the system’s roles and contracts, and they are only presented in Figure 5-3 and Figure 5-7.
5.3 Synthesis of the System Design Model from its Requirements

To ease the task of designing a context-aware adaptive system and maintain a casual connection between the system design and its requirements, we introduce a technique to synthesize the system’s design model from its requirements (specified as two sets of scenarios). The mappings between the system scenarios and its design model are summarised in Table 5-1. In general, the functional scenarios are mapped to the system’s functional composite, while the management composite is derived from the adaptation scenarios. The system properties are also transformed to constraints on the system’s functional contracts and behaviour processes. In recent years, a number of approaches (e.g. [150, 191-192]) have been introduced to synthesize design models of general software systems from their scenarios using a set of algorithms. Thus, similar to these approaches, we also introduce a set of algorithms to derive the design model of a context-aware adaptive system from its requirements (scenarios).

Table 5-1: The mappings between the system scenarios and its design model

<table>
<thead>
<tr>
<th>The System’s Scenarios</th>
<th>The System’s Design Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Scenarios</td>
<td>Functional Composite</td>
</tr>
<tr>
<td>Functional Participant</td>
<td>Functional Role and its Player</td>
</tr>
<tr>
<td>Contextual Participant</td>
<td>Context Role and its Context Provider</td>
</tr>
<tr>
<td>Functional Message</td>
<td>Functional Interaction</td>
</tr>
<tr>
<td>Set of Functional Messages</td>
<td>Functional Contract</td>
</tr>
<tr>
<td>Contextual Message</td>
<td>Context Attribute(s)</td>
</tr>
<tr>
<td>Set of Contextual Messages</td>
<td>Contextual Contract</td>
</tr>
<tr>
<td>State Invariant</td>
<td>Functional/Context Quality</td>
</tr>
<tr>
<td>Functional Scenario</td>
<td>Behaviour Process</td>
</tr>
<tr>
<td>Scenario’s First Message</td>
<td>Process’s Start Event</td>
</tr>
<tr>
<td>Scenario’s Last Message</td>
<td>Process’s End Event</td>
</tr>
<tr>
<td>Message (Functional or Contextual)</td>
<td>Task and its Completion Event</td>
</tr>
<tr>
<td>Sequence Reference</td>
<td>Composite Task and its Completion Event</td>
</tr>
<tr>
<td>Interaction Constraint</td>
<td>Task Pre-Condition</td>
</tr>
<tr>
<td>System Property</td>
<td>Conversation Clause/Process Constraint</td>
</tr>
<tr>
<td>Adaptation Scenarios</td>
<td>Management Composite</td>
</tr>
<tr>
<td>Management Participant</td>
<td>Organizer Role</td>
</tr>
<tr>
<td>Functional System Participant</td>
<td>Functional System Role</td>
</tr>
<tr>
<td>Management Message</td>
<td>Management Action(s)</td>
</tr>
<tr>
<td>Set of Management Message</td>
<td>Management Contract</td>
</tr>
<tr>
<td>Adaptation Fragment</td>
<td>Adaptation Rule(s)</td>
</tr>
</tbody>
</table>
5.3.1 Synthesis of the Functional Composite

To synthesize the system’s functional composite, the functional scenarios are used for deriving the composite elements as described below.

(A) **Functional Roles and their Players**: The functional roles capture the system functionality where each role description is an abstract definition of tasks that this role should perform. In the functional scenarios, the functional participants are involved in performing or requesting tasks. Thus, each functional participant is mapped to a functional role in the functional composite. For example, the user and the route planner roles in Figure 5-4 are corresponding to the functional participants “FL1” (the user) and “FL2” (the route planner) shown in Figure 4-6. In addition, a functional role needs to have one or more functional players to provide its actual functionality at runtime. However, the functional scenarios do not give enough information about how many players that a synthesized role should have. Therefore, a placeholder of a player is generated for each synthesised role and the software engineer can define more players later. For example, the engineer can specify two route planning algorithms that are able to play the route planner role as shown in Listing 5-1 (i.e. RoutePlanner1 and RoutePlanner2).

An algorithm to synthesizing a composite’s roles from a set of scenarios is shown in Listing 5-9. The algorithm maps each participant in the scenarios to a role (Line 5 in Listing 5-9). Then, if the role does not exist in the composite, the role is added to that composite (Lines 6 and 7 in Listing 5-9). Finally, a placeholder of a player for that role is created based on the role type and added to the composite. The player is also bound to the created role (Lines 8-10 in Listing 5-9). This algorithm can be used for synthesizing the functional roles of the functional composite from the functional scenarios by generating functional roles and players that are corresponding to the functional participations in the scenarios.

**Listing 5-9**: An algorithm (pseudocode) to synthesize a composite’s roles from a set of scenarios

```
1: void SynthesisoftheSystemRoles (Composite composite, SystemScenarios scenarios)
2:   FOR each Scenario S in scenarios
3:     ArrayList <Participant> participants= S.getParticipants();
4:     FOR each Participant PR in participants
5:       Role R = new Role (PR);
6:       IF R does not exists in composite THEN
7:         composite.addRole(R);
8:         Player PL = new Player (R.getType());
9:         composite.addPlayer(PL);
10:        Bind(PL, R);
11:     END IF
12:   END FOR
13: END FOR
14: END
```

(B) **Functional Contracts**: The functional contracts capture functional *interactions* between the system functional roles and their *qualities*. This corresponds to functional messages between
functional participants and their required level of qualities in the functional scenarios. As such, we traverse the scenarios and if there is any interaction between two functional participants, a functional contract is created between the functional roles corresponding to these participants (if the contract does not already exist) as presented in Listing 5-10 (Lines 2-10). For example, the contract “FC2” is formed between the user and the route planner roles as shown in Listing 5-1 because there are a set of interactions (functional messages) between the functional participants corresponding to the two roles (see Figure 4-6). Then, for each functional message, a functional interaction is created and added to the created (or existing) contract (if it does not already exist in the contract). Finally, if the functional message has a quality, this quality is added to the functional contract when it does not exist (Lines 11-17 in Listing 5-10). An example functional contract that can be synthesized from the interactions between the user and the route planner participants (presented in Figure 4-6) is “FC2” described in Listing 5-2.

Listing 5-10: An algorithm to synthesize a system’s functional contracts

```
1: void SynthesisofFunctionalContracts (FunctionalComposite FC, FunctionalScenarios FS)
2:     FOR each Scenario S in FS
3:         ArrayList <Message> messages= S.getMessages();
4:            FOR each Message M in messages
5:                IF M.getType() is "Functional" THEN
6:                    FunctionalContract contract=FC.getContract(M.getRoleA(), M.getRoleB());
7:                        IF contract does not exist in FC THEN
8:                           contract = new FunctionalContract(M.getRoleA(), M.getRoleB()) ;
9:                              FC.addFunctionalContract(contract);
10:                         END IF
11:                        FunctionalInteraction FI = new FunctionalInteraction(M);
12:                            IF FI does not exist in contract THEN
13:                                contract.addFunctionalInteraction(FI);
14:                           END IF
15:                       FunctionalQuality FQ = new FunctionalQuality (M.getQuality());
16:                           IF FQ does not exist in contract THEN
17:                               contract.addFunctionalQuality(FQ);
18:                          END IF
19:                     END IF
20:     END FOR
21: END FOR
22: END
```

(C) Context Roles and their Providers: Similar to the mapping the functional participants to functional roles, the contextual participants are mapped to context roles (e.g. the “TrafficInfo” participant presented in Figure 4-6 is mapped to the “TrafficInfo” role shown in Figure 5-4). To make the context information available, there is a need for context providers that monitor the context information. The context providers are part of the system’s solution space. Thus, like the functional roles’ players, they need to be specified by the software engineer and bound to the synthesized roles. For example, the engineer can specify two context providers for the traffic information role (e.g. road side unit and traffic information provider shown in Figure 5-4). The algorithm introduced in Listing 5-9 can be used for synthesizing the contextual roles and their providers from the scenarios, where each contextual participant is transformed to a context role bound to a context provider.
(D) **Contextual Contracts**: A contextual contract specifies context information that is needed by a functional role. In the scenarios, context information is captured by contextual messages. As such, we map a set of contextual messages between a contextual participant and a functional participant in the scenarios to a contextual contract in the functional composite. The algorithm for synthesizing the contextual contracts is similar to the algorithm used for synthesizing the functional contracts, and then algorithm details are omitted.

An example contextual contract that can be synthesized from the functional scenario “FS3” (shown in Figure 4-6) is “CC1” presented in Listing 5-3. This contract specifies that the route planner role needs to know the traffic information provided by the “TrafficInfo” role.

(E) **Behaviour Processes**: To synthesize the functional composite’s behaviour, each scenario is transformed to a behaviour process by the algorithm described in Listing 5-11. For example, the process “P3: PlanRoute” shown in Figure 5-6 corresponds to the functional scenario “FS3” presented in Figure 4-6.

In transforming a scenario to a process, the first interaction in the scenario defines the start event of the process. The process’s end event is the completion of the scenario’s last interaction. For example, the event “PlanRouteRequested” is the start event of the “PlanRoute” process, and the event “RouteSelected” is the process’s end event as shown in Listing 5-5. These two events are corresponding to the first and last messages of the scenario “FS3” shown in Figure 4-6. To identify such two events from a scenario, the type “start” is assigned to the first event of the generated process and the type “end” is assigned to the last event generated from the scenario (see Lines 7 and 11 in Listing 5-11).

Then, each interaction fragment in the scenario is transformed to a task or a number of tasks based on its type (Line 9 in Listing 5-11). In general, the fragment can be simple (i.e. a sequence reference or a message) or combined (e.g. an alternative fragment). Thus, based on the fragment type, part of the algorithm is used (i.e. add simple or combined fragment) as shown in Listing 5-11 (Lines 15-19). If an interaction is a message or a sequence reference, it is transformed to a single task. In addition, a type is assigned to this task. The type is either simple if it is a message or compound if it is a sequence reference (i.e. the task is another process itself). The task pre-condition is its previous interactions’ completion and its post-condition is an event that specifies the task completion to enable the execution of other dependent tasks as shown in Listing 5-11 (Lines 23-34). For example, the “ProvideRoute1” message shown in Figure 4-6 is transformed to “ProvideRoute1” task that has “SelectedAttractionsProvided and AttractionsSelected” as a pre-condition and “RoutesProvided” as a post condition as shown in Listing 5-5.
Listing 5-11: An algorithm to synthesize a system’s behaviour processes (Part 1)

```
1: void synthesisofaProcessModel(FunctionalScenario SM, ProcessModel PM)
2:     //FM: A System Scenario       AM: A System Process Model
3:     Event lastEvent = new Event();
4:     ArrayList<InteractionFragment> fragments = SM.getInteractionFragments();
5:     FOR each InteractionFragment interaction in fragments
6:         IF interaction is the first message THEN
7:             lastEvent.setType("Start");
8:         END IF
9:         lastEvent = AddFragment(interaction, PM, lastEvent);
10:     END FOR
11:     lastEvent.setType("End");
12: END
13: Event AddFragment(InteractionFragment interaction, ProcessModel PM, Event lastEvent)
14:     Event fragmentLastEvent = new Event();
15:     IF interaction.getType() is "Message" or "Reference" THEN
16:         fragmentLastEvent = AddSimpleFragment(interaction, PM, lastEvent)
17:     ELSE
18:         fragmentLastEvent = AddCombinedFragment(interaction, PM, lastEvent);
19:     END IF
20:     Return fragmentLastEvent;
21: END
22: Event AddSimpleFragment(InteractionFragment interaction, ProcessModel PM, Event lastEvent)
23:     Task T=new Task();
24:     T.setName(interaction.getName());
25:     IF interaction.getType() is "Reference" THEN
26:         T.setType("compound");
27:     END IF
28:     IF interaction has a constraint THEN
29:         Event e1 = new Event(interaction.getConstraint());
30:         T.addPreEvent(e1); PM.addEvent(e1);
31:     END IF
32:     T.addPreEvent(lastEvent);
33:     Event e2=new Event(); e2.setName(interaction.getName() + "Completed");
34:     PM.addEvent(e2); T.addPostEvent(e2); PM.addTask(T);
35:     Return e2;
36: END
```

In the case of an interaction is a combined fragment, a set of events and tasks are generated based on the combined fragment type by the algorithm shown in Listing 5-12. First, an optional fragment is translated to a set of events and tasks corresponding to the optional messages. These elements are then added to the process model as shown in Listing 5-12 (Lines 6-8). In addition, an event is generated to skip the optional messages when the condition of the fragment does not hold (Lines 4 and 9 in Listing 5-12). This event is also used as a pre-condition for the tasks to be executed after the optional fragment. An example transformed optional fragment using this part of the algorithm is shown in Figure 5-10.A.

Second, a loop fragment is transformed to a set of events and tasks that are corresponding to the interactions that need to be repeated as long as the loop condition still hold (see Lines 18-22 in Listing 5-12). In addition, to repeat the loop interactions, the pre-event for starting the loop is made as “completion of interactions before the loop and the loop condition is true (to execute the first iteration in the loop)” or “the last interaction of the loop is executed and the loop condition still hold (to execute another iteration of the loop)” as shown in Line 13 of Listing 5-12 and Figure 5-10.B. In the same manner, an event is generated to terminate the loop when
its condition is false after completing the last interaction (Line 14 in Listing 5-12). An example transformed loop fragment is shown in Figure 5-10.B.

**Listing 5-12:** An algorithm to synthesize a system’s behaviour processes (Part 2)

```java
1: Event AddCombinedFragment(InteractionFragment interaction, ProcessModel PM, Event lastEvent)
2:     IF type of the interaction is "Optional" THEN
3:         ArrayList <InteractionFragment> operands=interaction.getOperands();
4:         //an event to skip the optional fragments
5:         Event SkipOptional=new Event(operands.getPostEventOfLastInter() "OR" LastEvent "AND" interaction.getNotCondition());
6:         PM.addEvent(SkipOptional);
7:         FOR each InteractionFragment inter in operands
8:             lastEvent = AddFragment(inter, PM, lastEvent);
9:     END FOR
10:     ELSE IF type of the interaction is "Loop" THEN
11:         ArrayList <InteractionFragment> operands=interaction.getOperands();
12:         Task T=new Task(operands.getFirstInteraction());
13:         //Loop start and repeating event
14:         Event LStart=new Event(LastEvent "AND" interaction.getCondition "OR" operands.getPostEventOfLastInter() "AND" interaction.getCondition);
15:         Event LEnd=new Event(LastEvent "AND" interaction.getNotCondition "OR" operands.getPostEventOfLastInter() "AND" interaction.getNotCondition);
16:         PM.addEvent(LStart); PM.addEvent(LEnd);
17:         Event e2=new Event (T.getName() + ".completed"); lastEvent = e2;
18:         T.addPreEvent(LStart); T.addPostEvent(e2);PM.addTask(T);  
19:         FOR each InteractionFragment inter in operands
20:             IF inter is not the first interaction in the loop THEN
21:                 lastEvent = AddFragment(inter, PM, lastEvent);
22:             END IF
23:         END FOR
24:         lastEvent = LEnd;
25:     ELSE IF type of the interaction is "Alternative" THEN
26:         ArrayList <InteractionFragment> operandsIF=interaction.getOperandsIFPart();
27:         FOR each InteractionFragment inter in operandsIF {
28:             lastEvent = AddFragment(inter, PM, lastEvent);
29:         END FOR
30:         ArrayList <InteractionFragment> operandsELSE=interaction.getOperandsELSEPart();
31:         FOR each InteractionFragment inter1 in operandsELSE {
32:             lastEvent = AddFragment(inter1, PM, lastEvent);
33:         END FOR
34:         Event altEnd = new Event("EndAlternative"+ interaction.getId());
35:         PM.addEvent (altEnd);setAlternativePathsEnd (altEnd);
36:         lastEvent = altEnd;
37:     ELSE IF type of the interaction is "Parallel" THEN
38:         ArrayList <InteractionFragment> operands=interaction.getOperands();
39:         FOR each InteractionFragment inter in operands {
40:             AddFragment(inter, PM, lastEvent);
41:         END FOR
42:         Event ParEnd=new Event();setParallelEndEvent(ParEnd);
43:         PM.addEvent (ParEnd);
44:         lastEvent = ParEnd;
45:     END IF
46:     Return lastEvent;
47:END
```

Third, an alternative fragment is transformed to two execution paths in a generated process. The first path is executed when the fragment condition is true (i.e. the IF part of the fragment), while the second path is executed when the condition does not hold (i.e. the ELSE part of the fragment). The steps for generating elements of an alternative fragment are presented in Listing 5-12 (Lines 25-35). An example generated part of a process from an alternative fragment is shown in Figure 5-10.C
Finally, a parallel fragment is transformed to \( n \) execution paths in a behaviour process, where \( n \) is the number of parallel fragments (see the example shown in Figure 5-10.D). Each path has a number of tasks that need to be executed in parallel with other paths. In addition, the paths are enabled by the same event as shown in Lines 38-40 of Listing 5-12, so that they can be executed concurrently. Furthermore, an event is added to the process. This event specifies the end of the parallel execution by combining the end events of the parallel paths as an “and” condition (Line 41 of Listing 5-12).

(F) *Conversation Clauses of the Functional Contracts and the Temporal Constraints of the Behaviour Processes:* The system’s temporal properties that need to be preserved at runtime are captured graphically in a form similar to the scenarios as described in Section 4.1.4. These properties are corresponding to conversation clauses that are defined in the functional contracts and temporal constraints that are specified for the behaviour processes. Thus, we introduced an algorithm (shown in Listing 5-13) to transform the system properties to a set of conversation clauses and temporal constraints. The algorithm starts by parsing the scenarios that include the system properties. Then for each property a temporal constraint is created (see Lines 2-5 in Listing 5-13). The constraint is either added to a functional contract when it is related to binary interactions between two roles, or to a behaviour process when more than two participants are involved in defining the property (see Lines 6-12 in Listing 5-13). An example constraint that can be synthesized from the system properties shown in Figure 4-12 is the conversation clause “c1” described in Listing 5-2. This clause is corresponding to the property “PO2” presented in Figure 4-12.
5.3.2 Synthesis of the Management Composite

The management composite is responsible for adapting the system functionality in response to context changes, i.e., the same purpose of the adaptation scenarios. Thus, we use these scenarios to derive the composite elements. Below, we only discuss how to synthesize the management and functional system roles, the management player, and the management contract. The context roles and the contextual contracts can be synthesized as discussed above as they are common between the functional and the management composites.

(A) The Management Role: A management participant in the adaptation scenarios has the same purpose of a management role in the management composite. Therefore, the management participant in the adaptation scenarios is mapped to the management role of the system’s design model. For example, the organizer role which corresponds to the organizer participant in Figure 4-8 and Figure 4-9 is added to the management composite shown in Figure 5-8.

(B) The Functional System Role: In the adaptation scenarios, a functional system participant represents the functional scenarios that need to be adapted at runtime. Similarly, in the system design, a functional system role represents the functional composite that needs to be adapted in response to context changes (and can be derived from the functional scenarios). Because of this correspondence, the functional system participant is transformed to the functional system role that is played by the system’s functional composite. For example, the system functionality role in Listing 5-6 is derived from the travel guide system participant shown in Figure 4-8.

(C) The Management Player: To decide the system adaptation in response to runtime context changes, we model the system’s management player as a set of Event-Condition-Action rules. Thus, we introduced an algorithm (shown in Listing 5-14) to synthesize the adaptation rules of the management player from the adaptation scenarios. In general, we map a contextual message
and an adaptation fragment triggered by this message to an adaptation rule when the fragment
does not have an ELSE part, while they are transformed to two rules otherwise because an ECA
rule does not have an ELSE part (Line 15 in Listing 5-14). First, the contextual message is used
for specifying the event that triggers the rule execution (Line 5 in Listing 5-14). For example,
the contextual message “CM1” presented in Figure 4-9 is transformed to the event of the rule
“TrafficInfoNotAvailable” specified in Listing 5-7. Second, the rule condition is same as the
adaptation condition when the IF part of the adaptation fragment is transformed to a rule (see
the correspondence between the adaptation conditions in Figure 4-9 and Listing 5-7), while the
negation of the condition is used in generating a rule corresponds to the fragment’s ELSE part
(Line 16 in Listing 5-14). Third, the rule’s adaptation actions are corresponding to management
messages defined in the adaptation fragments. Thus, the management messages in the scenarios
are transformed to adaptation actions in the system’s design model (Lines 10 and 20 in Listing
5-14). The correspondences between the management messages and the adaptation actions are
summarised in Table 5-2.

Listing 5-14: An algorithm to synthesize a system’s adaptation rules and management contract

Example rules that can be synthesized from the adaptation fragments presented in Figure 4-9
are the rules shown in Listing 5-7. The rule “TrafficInfoNotAvailable” can be derived from the
fragment “AD1”. First, the contextual message “CM1: NotifyContext (TrafficInfoAvailability)”
is transformed to the event “ValueChanges (TrafficInfoAvailability)”. Second, the condition of
“AD1” is same as the adaptation condition of the rule “TrafficInfoNotAvailable”. Third, using
Table 5-2, a set of adaptation actions that are corresponding to the management messages can be generated. For example, the actions:

“[RemoveTaskFromProcess (“P3”, “GetTrafficInfo”) and
RemoveEventFromProcess (“P3”, “TrafficInfoAvailable”)]”
can be synthesized from the management message “MM1: RemoveMessage (FS3.CM1)”. In the same manner, the fragment “AD2” can be transformed to the rule “TrafficInfoAvailable”.

**Table 5-2:** The mappings between the management messages and the adaptation actions

<table>
<thead>
<tr>
<th>Management Message</th>
<th>Adaptation (Management) Action(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLifeLine</td>
<td>- AddRole</td>
</tr>
<tr>
<td>RemoveLifeLine</td>
<td>- RemoveRole, RemoveFunctionalContract, RemoveContextualContract</td>
</tr>
<tr>
<td>ChangeMessageParameters</td>
<td>- ChangeInteractionParameters OR AddContextAttributeToContract, RemoveContextAttributeFromContract</td>
</tr>
<tr>
<td>ChangeMessageName</td>
<td>- ChangeInteractionName</td>
</tr>
<tr>
<td>ChangeMessageReturn</td>
<td>- ChangeInteractionReturn</td>
</tr>
<tr>
<td>AddStateInvariant</td>
<td>- AddFunctionalQuality OR AddContextQuality</td>
</tr>
<tr>
<td>RemoveStateInvariant</td>
<td>- RemoveFunctionalQuality OR RemoveContextQuality</td>
</tr>
<tr>
<td>ChangeStateInvariant</td>
<td>- ChangeFunctionalQuality OR ChangeContextQuality</td>
</tr>
<tr>
<td>AddFunctionalProperty</td>
<td>- AddConversationClause OR AddTemporalConstraint</td>
</tr>
<tr>
<td>RemoveFunctionalProperty</td>
<td>- RemoveConversationClause OR RemoveTemporalConstraint</td>
</tr>
<tr>
<td>ChangeFunctionalProperty</td>
<td>- ChangeConversationClause OR ChangeTemporalConstraint</td>
</tr>
<tr>
<td>ChangeSequenceReference</td>
<td>- ChangeTask</td>
</tr>
<tr>
<td>AddCombinedFragment</td>
<td>- AddEventToProcess, ChangeTaskPreEvent, ChangeTaskPostEvent</td>
</tr>
<tr>
<td>RemoveCombinedFragment</td>
<td>- RemoveEventFromProcess, ChangeTaskPreEvent, ChangeTaskPostEvent</td>
</tr>
<tr>
<td>ChangeFragmentCondition</td>
<td>- ChangeEvent</td>
</tr>
<tr>
<td>AddMessageToFragment</td>
<td>- ChangeTaskPreEvent</td>
</tr>
<tr>
<td>RemoveMessageFromFragment</td>
<td>- ChangeTaskPreEvent</td>
</tr>
</tbody>
</table>

(D) *Management Contract:* To apply the adaptation rules’ actions, the system that needs to be adapted should support the application of these actions. To do so, a management contract between the functional system role and the organizer role is formed (e.g. the contract “MC1” in Figure 5-8). This contract defines adaptation actions that need to be performed on the functional system at runtime. To synthesize the management contract, the adaptation scenarios are parsed and the adaptation actions to be performed on the functional system are identified and added to the management contract as shown in Listing 5-14 (Lines 11 and 21). The management contract
also contains a set of states that need to be transferred when the system is adapted in response to context changes. For example, the active events in the process “P3” need to be transferred when this process is adapted at runtime as defined by the state transfer “st1” presented in Listing 5-8. The state transfers are based on the system design, and then they need to be specified manually by the software engineer.

5.4 Realization of Context-aware Adaptive Software Systems

To realize a context-aware adaptive software system designed in our approach, we have used the ROAD framework [62], where it follows the organizational approach as our approach does. This framework is an extension to the Apache Axis2 to realize adaptive software systems [47]. To use the ROAD framework, we transform a system’s model designed in our approach to an executable model that is compatible with the ROAD framework. In the following, we describe the ROAD model briefly (more details about the ROAD model can be found in [45, 47, 140]), and the transformation from our design model to the executable ROAD model to enable the system realization.

5.4.1 Background: Modelling Adaptive Software Systems in the ROAD Approach

Following the organizational approach, the ROAD approach specifies an adaptive system as a composite (see Listing 5-15) [62]. This composite has a set of roles that interact with each other by contracts to provide the system functionality. It also has a set of players to play the roles and a set of facts that provide information needed by these roles. Furthermore, the composite has a set of processes to capture the system behaviour, and an organizer role that can be played by a human or a computer program to adapt the system in response to context changes.

(A) Roles and their Players: The composite has a number of roles. Each role description is an abstract definition of tasks that this role should provide (i.e. role interface). For example, the route planner role is responsible for performing the “PlanRoute” task as shown in Listing 5-15 (Line 15). In addition, a role can provide or consume some information. Thus, each role has a set of linked facts. For example, the route planner role needs the traffic information, and then it has the fact “TrafficInfo” as a linked fact (see Line 19 in Listing 5-15, i.e., consume="true" provide="false"). This fact is provided by the “TrafficInfo” role as shown in Listing 5-15 (Line 27). Furthermore, each role has one or more players to perform (carry out) its tasks at runtime (i.e. role’s implementations). For example, the player “RoutePlanner1” can be bound to the route planner role to perform its tasks (Lines 37-39 in Listing 5-15). The system players may already exist and need to be bound to the roles, or they are new and need to be developed.
Listing 5-15: Part of the XML description for the travel guide system following the ROAD approach

```
<xml version="1.0" encoding="UTF-8">
  <tns:SMC name="TravelGuideSystem">
    <!-- Facts -->
    <Facts>
      <Fact name="TrafficInfo">
        <Attributes>
          <Attribute>TrafficInfo</Attribute>
        </Attributes>
      </Fact>
      <Fact name="ContextInfoAvailability">
        ...
      </Fact>
    </Facts>
    <!-- Composite Roles -->
    <Roles>
      <Role id="FR3" name="RoutePlanner">
        <Tasks>
          <Task name="PlanRoute"/>
          <Task name="SelectRoute"/>
          ...
        </Tasks>
        <LinkedFacts>
          <Fact consume="true" provide="false" name="TrafficInfo"/>
        </LinkedFacts>
      </Role>
      <Role id="CR3" name="TrafficInfo">
        <Tasks>
          <Task name="GetTrafficInfo"/>
          ...
        </Tasks>
        <LinkedFacts>
          <Fact consume="false" provide="true" name="TrafficInfo"/>
        </LinkedFacts>
      </Role>
      <Role id="OR1" name="Organizer">
        <LinkedFacts>
          <Fact consume="true" provide="false" name="ContextInfoAvailability"/>
        </LinkedFacts>
      </Role>
    </Roles>
    <!-- Role Players' Binding -->
    <PlayerBindings>
      <PlayerBinding playerid="FP3" roleid="FR3">
        <Endpoint> http://localhost:8080/axis2/services/RoutePlanner </Endpoint>
      </PlayerBinding>
      <PlayerBinding playerid="FP1" roleid="FR3">
        <Endpoint> http://localhost:8080/axis2/services/RoutePlanner </Endpoint>
      </PlayerBinding>
      <PlayerBinding playerid="FP1" roleid="CR3">
        <Endpoint> http://localhost:8080/axis2/services/TrafficInfo </Endpoint>
      </PlayerBinding>
    </PlayerBindings>
    <!-- Composite Contracts -->
    <Contracts> ...
    </Contracts>
    <!-- Process Definitions -->
    <ProcessDefinitions> ...
    </ProcessDefinitions>
  </tns:SMC>
</xml>
```

(B) Contracts: The contracts specify and manage interactions between the system roles. Each contract has a number of items. First, the contract has an identifier and is formed between two roles. For example, the contract “FC2” is formed between the user and the route planner roles as shown in Listing 5-16 (Lines 23 and 24). Second, the contract has a set of terms (interactions) between the contracted roles as shown in Listing 5-16. Each interaction has: (1) an identifier (e.g. “FC2-I1”); (2) an operation that has a name (e.g. “PlanRoute”) and a set of parameters (e.g. 
current location and destination) as shown in Lines 8-17 of Listing 5-16, and it is performed when the interaction is requested; (3) a direction to specify who is responsible for performing the operation defined in the interaction (e.g. “AtoB” that means the route planner is responsible for providing the route calculation operation); (4) a return type (e.g. void) as shown in Line 18 of Listing 5-16.

**Listing 5-16:** An example contract in the travel guide system

```
<xml version="1.0" encoding="UTF-8">
<tns:SMC name="TravelGuideSystem">
<!-- Composite Contracts -->
<Contracts>
  <Contract id="FC2" ruleFile="TravelGuideSystem_FC2.drl">
    <Terms>
      <Term id="FC2-I1" name="PlanRoute">
        <Operation name="PlanRoute">
          <Parameters>
            <Parameter>
              <Type>String</Type>   <Name>Location</Name>
            </Parameter>
            <Parameter>
              <Type>String</Type>   <Name>Destination</Name>
            </Parameter>
          </Parameters>
          <Return>void</Return>
        </Operation>
      </Term>
    </Terms>
  </Contract>
...  
</Contracts>
</tns:SMC>
```

(C) **Behaviour Processes:** In ROAD model, the system behaviour is defined as event-based processes [140]. A process consists of a start event to initiate its execution, an end event to terminate the process, and a set of behavior units. A behavior unit is a collection of tasks that are operations executable by the system’s role players. Each task has a pre-condition (events that enable the execution of a task) and a post-condition (events that are generated upon a task completion for enabling the execution of other dependent tasks). For example, the “plan a route” process shown in Listing 5-17 has the event “PlanRouteRequested” as the start event (Line 6), and the event “RouteSelected” as the end event (Line 7). It also has a set of behaviour units such as “PlanRoute” (Line 9 in Listing 5-17). This unit has a reference to the task “FR3.PlanRoute” (Line 27 in Listing 5-17) with the pre-condition “PlanRouteRequested” and the post-condition “CalculateRoutes”. In general, this process starts by a user’s request to plan a route. Then, the live traffic information and the user’s selected attractions are acquired. Based on the selected attractions, a suitable route planning function is selected where “ProvideRoutes1” is used when a set of attractions are selected by the user and “ProvideRoutes2” is used otherwise. Finally, a set of routes are suggested to the user, where he can select a route and start his journey.
Listing 5-17: An XML description of a process in the travel guide system by the ROAD approach

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tns:SMC name="TravelGuideSystem">

<!-- Process Definitions -->
<ProcessDefinitions>

<!-- Process Definitions -->
<tns1:ProcessDefinition id="P3" description="Process for Plan a Route">

<tns1:TaskRefs>
</tns1:TaskRefs>
</tns1:ProcessDefinition>

</ProcessDefinitions>

</tns:SMC>
```

To trigger events that enable a task execution or that indicate a task completion, the system contracts have a set of rules (specified in Drools\(^6\) format) that are fired when an interaction is requested. Example rules are shown in Listing 5-18. These rules are part of the contract “FC2” (see Line 5 in Listing 5-16). When an interaction is requested through this contract, the contract rules are evaluated and events are generated based on the interaction name. In Listing 5-18, the event “PlanRouteRequested” is generated when the “PlanRoute” interaction is requested (Lines 4-9 in Listing 5-18), while the event “CalculateRoutes” is triggered upon the completion of the request of the “PlanRoute” task (Lines 10-15 in Listing 5-18).

(D) **Organizer Role and its Player:** The system composite has an organizer role. This role has the ability to adapt the system functionality at runtime by a set of standard methods that are engineered into it. An example action that can be executed by this role is addNewRole (“FR4”, “AttractionsLocator”, “The role represents the attractions finder service”). In addition, the role is able to monitor information that triggers the system adaptation (e.g. the traffic information availability as shown in Line 32 of Listing 5-15). Furthermore, to decide adaptation actions that need to be applied on the system, the organizer role has an organizer player that is responsible for this task (as shown in Line 43 of Listing 5-15). This player can be a graphical user interface

\(^6\) [http://www.jboss.org/drools](http://www.jboss.org/drools)
that enables the software engineer to adapt the system manually, or a program that automatically
decides a set of adaptation actions that need to be applied on the system in response to runtime
context changes.

**Listing 5-18:** An example rule file for the contract “FC2”

```java
2: declare MessageRecievedEvent
3: @role(event)
4: /** Rules **/
5: rule "PlanRoute"
6:    when
7:       $event:MessageRecievedEvent(operationName == "PlanRoute", response == false)
8:    then
9:       $event.triggerEvent("PlanRouteRequested");
10: end
11: rule "PlanRouteResponse"
12:   when
13:   $event : MessageRecievedEvent(operationName == "PlanRoute", response == true)
14: then
15:   $event.triggerEvent("CalculateRoutes");
16: end
```

5.4.2 Transforming our Model to an Executable ROAD Model

ROAD and our approaches are similar because they follow an organizational approach. This
similarity enables us to realize context-aware adaptive systems using the ROAD framework by
transforming a system model designed in our approach to an executable ROAD model. Table
5-3 summarizes the mappings between our model and the executable model, and we describe
these mapping in detail below.

**Table 5-3:** Mappings between a system model designed in our approach and the ROAD model

<table>
<thead>
<tr>
<th>Our Design Model</th>
<th>ROAD Executable Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional and Management Composites</td>
<td>An executable ROAD Composite</td>
</tr>
<tr>
<td>Functional or Context Role</td>
<td>Role</td>
</tr>
<tr>
<td>Functional or Contextual Contract in the Functional Composite</td>
<td>Contract and a set of Tasks</td>
</tr>
<tr>
<td>Process</td>
<td>Process</td>
</tr>
<tr>
<td>Start and End Events</td>
<td>Start and End Events</td>
</tr>
<tr>
<td>Task</td>
<td>Behaviour unit with one Task and files for the rules that trigger a process’s events</td>
</tr>
<tr>
<td>Organizer Role</td>
<td>Organizer Role</td>
</tr>
<tr>
<td>Contextual Contract in the Functional and Management Composite</td>
<td>Fact with a set of context attributes</td>
</tr>
<tr>
<td>Adaptation Rules</td>
<td>We automatically generate an organizer player</td>
</tr>
<tr>
<td>Management Contract</td>
<td>Adaptation actions that are supported by the organizer role</td>
</tr>
<tr>
<td>Functional System Role</td>
<td>Not needed where there is only one system composite</td>
</tr>
<tr>
<td>Functional Player and Context Providers</td>
<td>We generate a player or a context provider that has a set of methods with empty bodies</td>
</tr>
</tbody>
</table>
To derive an executable ROAD composite that is corresponding to a system’s functional and management composites, a number of transformations are performed to generate the composite structure, the behaviour processes, and the system’s players and context providers as shown in Listing 5-19.

(A) Generating the ROAD Composite Structure: First, to represent the concept of contextual contracts in the executable ROAD model, the context information is modelled as a set of facts. In our model, the contextual contracts are used to derive context roles’ descriptions, and then a context role can be seen as a collection of context attributes. This establishes a correspondence between a fact in ROAD terms and a context role in our model. Thus, we use the contextual contracts to derive context roles, and then we transform each context role description to a fact in the executable ROAD model. An example fact that can be derived from the travel guide model is the “TrafficInfo” fact shown in Listing 5-15 (Lines 4-8). This fact is corresponding to the contextual contract “CC1” shown in Figure 5-4.

Listing 5-19: An algorithm to generate an executable ROAD model from a system’s design model

<table>
<thead>
<tr>
<th>ROADModel GenerateROADModel(FunctionalComposite FC, ManagementComposite MC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ROADM odel RM = new ROADM odel();</td>
</tr>
<tr>
<td>2. GenerateCompositeStructure(RM, FC, MC);</td>
</tr>
<tr>
<td>3. GenerateCompositeProcesses(RM, FC);</td>
</tr>
<tr>
<td>4. GenerateCompositePlayers(RM, FC, MC);</td>
</tr>
<tr>
<td>5. Return RM;</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void GenerateCompositeStructure (ROADModel RM, FunctionalComposite FC, ManagementComposite MC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. AddFacts (RM, FC, MC);</td>
</tr>
<tr>
<td>7. AddContracts(RM, FC);</td>
</tr>
<tr>
<td>8. AddRoles(RM, FC, MC);</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void GenerateCompositePlayers (ROADModel RM, FunctionalComposite FC, ManagementComposite MC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. GenerateFunctionalPlayers(RM, FC);</td>
</tr>
<tr>
<td>10. GenerateContextProviders(RM, FC, MC);</td>
</tr>
<tr>
<td>11. GenerateOrganizerPlayer (RM, MC);</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

Second, the functional and contextual contracts of the functional composite are translated to ROAD contracts. The functional contracts are translated to ROAD contracts in a straightforward manner, where they have the same elements. For example, the functional contract “FC2” shown in Listing 5-16 can be generated from the contract “FC2” shown in Listing 5-2. The contextual contracts are transformed to ROAD contracts by transforming each context attribute to two interaction terms. The first term enables the context role to notify the context changes, while the second term enables the functional role connected to this contract to get the context attribute when needed. An example transformed contextual contract is presented in Listing 5-20. It can be derived from the contract “CC1” shown in Listing 5-3 by transforming the context attribute
“TrafficInfo” to two interactions “GetTrafficInfo” and “NotifyTrafficInfo”. During the contracts generation, we also identify tasks to be carried out by the roles according to the directions of the interaction terms (as discussed above). For each interaction, a task is added to one of the roles connected to this contract. For example, the “PlanRoute” task is added to the route planner role, where this role is responsible for doing that task as shown in the description of the route planner role (Line 15 of Listing 5-15).

**Listing 5-20:** An example transformed contextual contract in the travel guide system

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tns:SMC name="TravelGuideSystem">

<!-- Composite Contracts -->
<Contracts>

<Contract id="CC1" ruleFile="TravelGuideSystem_CC1.drl">

<Terms>

<Term id="CC1-I1" name="GetTrafficInfo">
<Operation name="GetTrafficInfo">
<Parameters></Parameters>
<Return>String</Return>
</Operation>
<Direction>AtoB</Direction>
</Term>

<Term id="CC1-I2" name="NotifyTrafficInfo">
<Operation name="NotifyTrafficInfo">
<Parameters>
<Parameter><Type>String</Type><Name>TrafficInfo</Name></Parameter>
</Parameters>
<Return>void</Return>
</Operation>
<Direction>BtoA</Direction>
</Term>

</Terms>

<RoleA>RoutePlanner</RoleA>
<RoleB>TrafficInfo</RoleB>

</Contract>

...</Contracts>
</tns:SMC>
```

Third, in our design model, we have a management contract to specify adaptation actions to be applied on the system functionality. However, the concept of the management contract is not considered in the ROAD approach. In ROAD model, the organizer role is engineered with a set of adaptation methods to enable the system adaptation in response to context changes. As such, there are correspondences between the actions of the management contract and the methods of the organizer role. Consequently, to realize the management contract, we use these methods to apply the adaptation actions at runtime.

Fourth, in ROAD, the system is modelled as one composite, and then the functional system role is not needed anymore. Therefore, in deriving the roles of the ROAD composite, all roles of the functional and management composites except the functional system role are transformed to ROAD roles. In addition, during the generation of a role, two items are added to the role: linked facts and tasks (if any). The linked facts are identified from the contextual contracts connected
with this role. For each contract, a fact corresponds to the context role that is connected to this contract is added to the role. For example, the route planner role shown in Listing 5-15 contains the linked fact “TrafficInfo” because of the contextual contract “CC1” shown in Figure 5-4. The tasks that need to be performed by this role (identified during the contracts generation) are also added to the role (e.g. the “PlanRoute” task is added to the route planner role).

(B) Generating the ROAD Behaviour Processes: An executable ROAD process (see Section 5.4.1) is an event-based process which is similar to our process model. Thus, each behaviour process in our design model is transformed directly into an executable ROAD process. But, a process designed in our approach is a collection of tasks, while a ROAD process is captured as a set of behaviour units and each unit consists of a set of tasks. Thus, we transform each task in our model to a behavior unit that contains a single task. An example ROAD process that can be generated from “PlanRoute” process shown in Figure 5-6 is shown in Listing 5-17, where each task is translated to a behaviour unit (e.g. the task “PlanRoute” is transformed to the behaviour unit “PlanRoute” that has a single task “FR3.PlanRoute”). To trigger the process’s events, we generate a set of Drools rules. For each task, we add two rules to the contract that contains the interaction term corresponds to this task. The first rule is fired when the task (the interaction) is requested, while the second rule is fired when the task execution is completed to generate events upon the task completion (see the example Drools rules shown in Listing 5-18).

(C) Generating the System Players: To have a fully functioning system, a set of players and context providers need to be developed and bound with the system roles. To ease this task, we automatically generate parts of the functional players and the context providers, and a fully working organizer player. First, a functional player provides a set of functions (tasks). As such, using the system’s design model, we generate a Java class (corresponding to the player) with empty methods that need to be completed by the system developers. An example Java code for the skeleton of the route planner player is presented in Listing 5-21. In the same manner, a set of Java classes are generated for the context providers. These classes’ skeletons contain methods that need to be implemented to provide the context information.

Listing 5-21: A Java code corresponding to the functional player “RoutePlanner1”

```java
public class RoutePlanner1 {
    public void PlanRoute(String Location, String Destination){
        System.out.println(" Executing the Operation 'PlanRoute' ");
        //To Do
    }

    public void NotifyTrafficInfo (String TrafficInfo){
        System.out.println(" Executing the Operation 'NotifyTrafficInfo' ");
        //To Do
    }
    ...
}
```
A set of functional players and context providers may already exist and provide the tasks and context information needed by the system. To use these players and providers during the system development, the system developers need to write a small amount of code that calls the already implemented methods instead of implementing the generated empty methods.

Second, to enable the execution of the *adaptation rules*, we transform them to Drools rules, so that the Drools rule engine can be used for their execution to decide the required adaptation in response to runtime context changes. In transforming a rule, we use the Drools rule “When” part to specify the rule event (e.g. changes in the status of the traffic information availability as shown in Listing 5-22). In addition, the rule “Then” part is used to capture the rule’s condition and actions. To evaluate a rule condition, we created a Java class called “ConditionEvaluator” that includes a method called “evaluate”. This method takes a condition as an input, replaces the condition variables with current context values, and returns true or false based on the condition evaluation. If the condition is true (e.g. the traffic information is not available), a number of adaptation actions are triggered (e.g. “RemoveContract_CC1” and “RemoveRole_TrafficInfo” as shown in Listing 5-22). The adaptation rule “R1” shown in Listing 5-22 is generated from the “TrafficInfoAvailable” rule described in Listing 5-7.

**Listing 5-22**: An example adaptation rule in the Drools format

```java
import au.edu.swin.ict.CAAS.CAASManagement.ContextEvent;
import au.edu.swin.ict.CAAS.CAASManagement.AdaptationActions;
import au.edu.swin.ict.CAAS.CAASManagement.ConditionEvaluator;

/** Global Variables **/
global AdaptationActions actions; // a variable to hold the triggered actions
global ConditionEvaluator conditionEvaluator;

rule "R1"
when
  ContextEvent( name == "TrafficInfoAvailability_Value_Chaged" )
then
  if(conditionEvaluator.evulate("(TrafficInfoAvailability == False)")){
    actions.addAction("RemoveContract_CC1");
    actions.addAction("RemoveRole_TrafficInfo");
    actions.addAction("Bind_RoutePlanner_RoutePlanner2");
    actions.addAction("RemoveTask_P3_GetTrafficInfo");
    actions.addAction("RemoveEvent_P3_TrafficInfoAvailable");
    actions.addAction("ChangeTaskPreEvent_P3_t3_CalculateRoutes");
  }
end
```

To use the transformed adaptation rules to decide adaptation actions in response to context changes, we automatically generate a Java class that represents the *organizer player*. This class has two methods: `decide` and `act` the adaptation actions. The method “decideAdaptationActions” shown in Listing 5-23 decides the adaptation actions using the Drools rule engine. When a set of events (context changes) are triggered, they are inserted into the rule engine to fire the relevant adaptation rules (Lines 6-7 in Listing 5-23). Then, the fired actions are retrieved from the rule
engine (Line 9 in Listing 5-23). For each action, checking is done to ensure that it does not exist in the accumulated list of actions (Lines 11-14 in Listing 5-23). If the action is new, it is added to the actions list (Line 16 in Listing 5-23). The result of this process is a set of actions in a form of an “adaptation script”. To take the dependences between the script actions into account (e.g. a role cannot be removed before removing contracts that it is involved in), the script actions are automatically ordered as shown in Line 22 of Listing 5-23.

Listing 5-23: Deciding the adaptation actions in response to context changes

```java
public ArrayList<String> decideAdaptationActions (){ 
    ArrayList<String> requiredActions=new ArrayList<String> ();
    try { 
        for(int j=0;j<events.size();j++){
            AdaptationActions a=new AdaptationActions();
            ksession.setGlobal("actions", a);
            ContextEvent e=new ContextEvent(); e.setName((String)events.get(j));
            ksession.insert(e); ksession.fireAllRules();
            for(int i=0;i<a.getAdaptionActionsSize();i++){
                String act=a.getActions(i);
                boolean added=false;
                for(int k=0;k<requiredActions.size();k++){
                    if(act.equals((String)requiredActions.get(k)))
                    {added= true; }
                }
                if(!added){
                    requiredActions.add(a.getActions(i)); }
            }
        }
    }catch (Throwable t) {
        t.printStackTrace();
    }
    OrderScript(requiredActions);
    return requiredActions;
}
```

In response to a context change (triggered as events), the method “decideAdaptationActions” generates an adaptation script. To execute this script, the method “actAdaptationActions” shown in Listing 5-24 adapts the running system by invoking adaptation methods of the organizer role that are corresponding to the script actions. This method has a reference to the running system’s organizer role (Line 2 in Listing 5-24). In addition, some details from the running system model are needed. For example, to add a role, there is a need for its name, identifier, and description. Therefore, the act method uses the `models@runtime` concept [135] to maintain an instance of the system’s executable model as shown in Lines 5-7 of Listing 5-24 (i.e. runtime representation of the system model). To execute a script action, the action type and the element to carry out the action on are identified (Lines 9-15 in Listing 5-17). Then, based on the action type, a method of the system’s organizer role is executed. For example, when an action type is “RemoveRole”, the “removeRole” method of the organizer role is called (Line 17 in Listing 5-24). In the case of some details are needed from the system model, the organizer player parses the runtime model to get the needed details and an adaptation method is executed with such details. For example, to add a role, the role details are acquired from the runtime model. Then, the “addNewRole”
method is executed with the role details (Lines 19-20 in Listing 5-24). Thus, using this method, adaptation actions decided by the “decideAdaptationActions” method presented in Listing 5-23 in response to a context change can be applied to the running system automatically.

Listing 5-24: Acting the adaptation actions that are generated in response to context changes

```java
public void actAdaptationActions (ArrayList<String> actions){
  try{
  Tvg_organizerStub organizer = new Tvg_organizerStub("http://localhost:8080/axis2/
  services/tvg_organizer/");
  organizer._getServiceClient().getOptions().setSoapVersionURI(org.apache.axiom.
  soap.SOAP11Constants.SOAP_ENVELOPE_NAMESPACE_URI);
  organizer._getServiceClient().getOptions().setTimeOutInMilliSeconds(400000);
  File file= new File("TVGSystem.xml");
  CompositeDemarsheller demarsheller = new CompositeDemarsheller();
  Composite TVGComposite = demarsheller.demarshalSMC(file.getAbsolutePath());
  for(int i=0;i<actions.size();i++){
    StringTokenizer st= new StringTokenizer(actions.get(i), " ");
    String actionType=st.nextToken(); //Action Type
    String actionElement=st.nextToken(); //Element to perform the action on
    String actionElement1= ""; //Element to be change inside a composite element
    if(st.hasMoreTokens()){
      actionElement1=st.nextToken();
    }
    if(actionType.equals("RemoveRole")){
      organizer.removeRole(actionElement);
    }
    else if(actionType.equals("AddRole")){
      Role role=(Role)TVGComposite.getRoleByID(actionElement);
      organizer.addNewRole(role.getId(), role.getName(), role.getDescription());
    }
    else if(actionType.equals("RemoveContract")){
      ...
    }
  }catch(Exception e){
    e.printStackTrace();
  }
}
```

5.4.3 Creating a Fully Functioning Software System

The transformation process described in Section 5.4.2 generates an executable model that is compatible with the ROAD framework (the details of the executable model for the travel guide system are presented in Appendix “A”). It also generates a fully working organizer player and skeletons for the functional players and the context providers. To make the travel guide system fully functioning, we have completed the code required in the generated functional players (e.g. the graphical user interface, i.e., SystemGUI), and context providers (e.g. the user information provider). We also used Google maps services\(^7\) to develop some of the functional players such as the route planners and the attractions finder.

When the executable model and the system’s players and context providers are deployed to the ROAD runtime environment, an instance of the system is created. This instance contains the system’s roles, players, contracts, and behaviour processes. It also supports the application of different adaptation actions to the system while in operation. For example, the deployed system

\(^7\) [http://code.google.com/apis/maps/documentation/webservices/](http://code.google.com/apis/maps/documentation/webservices/)
composition is able to add, remove, or modify its roles and contracts (i.e. an adaptable system composition). Similarly, each deployed system artefact (e.g. a contract) has the ability to add, remove, and modify its sub-elements (e.g. adding a functional interaction). Thus, the system’s executable artefacts are “adaptable artefacts”.

An example runtime adaptation of the travel guide system is shown in Figure 5-11. In Figure 5-11.A, the system only includes the attractions finder service that suggests to the tourist a set of attractions based on his current location and the weather condition. He can select some of them to visit. To plan a route to explore the attractions, the tourist requests the route planning service to be included in his system. Thus, the system is adapted to include such service by performing a set of adaptation actions such as adding the route planner role, the traffic information role, the contracts “FC2” and “CC1”, and the behaviour process “P3”. When this service is included and ready, it acquires the tourist location, his attractions list, and the traffic information to suggest a suitable route (see Figure 5-11.B).

Figure 5-11: An example runtime adaptation of the travel guide software system

5.5 Summary

In this chapter, we have first introduced a meta-model to enable the design of context-aware adaptive systems. Following this meta-model, a system is modelled as two main composites: functional and management. The functional composite captures the system functionality, while the management composite specifies the adaptations to the functional composite in response to context changes. To design the system composites, we adopt an organizational approach where a composite’s elements and their relationships are explicitly represented. As such, the composite elements and their relationships can be clearly captured. In addition, this approach keeps the composite structure alive at runtime, so that the composite can be easily manipulated (adapted) in response to runtime context changes.
Second, to ease the system design and to maintain a casual connection between the system’s design and its requirements, we have introduced a technique to *synthesize* the system’s initial design from its requirements (specified as functional and adaptation scenarios). In general, the functional scenarios are mapped to the system’s functional composite, while the management composite is derived from the adaptation scenarios.

Third, to *realize* a context-aware adaptive software system designed in our approach, we used the ROAD framework. To use this framework, we transform the system’s design model to an executable model that is compatible with the ROAD framework. When the generated model is deployed to the ROAD runtime environment, an instance of the software system is created. This instance supports the application of different adaptation actions to the system at runtime.

The contribution of this chapter is twofold: (1) an *organization-based meta-model* to enable the design of context-aware adaptive software systems; (2) techniques to *synthesize* a system’s *design model* from its requirements (specified as a set of scenarios), and to *transform* the design model to an *executable ROAD model* for the deployment.
Chapter 6: Runtime Evolution of Context-aware Adaptive Software Systems

During its lifetime, a context-aware adaptive software system needs to evolve in response to unanticipated changes [5, 193]. These changes can be in its environment or requirements where (a) the system is deployed into an environment that is not totally anticipated at the development time; (b) the users may need a new functionality, or the provider wants to enhance the software system with a new feature. In addition, the provider wants to reduce the downtime of the system to increase the users’ satisfaction [8, 66]. Therefore, such context-aware adaptive system needs to have the ability to evolve at runtime in response to the unanticipated changes.

As discussed in Section 3.5, to support the runtime evolution of a context-aware adaptive software system in response to unanticipated changes, existing approaches allow the software engineer to change the system’s requirements specification and its design model in response to the unanticipated changes in an ad-hoc manner (e.g. [8, 29]). In addition, to validate changes to the system requirements by the existing techniques (e.g. [23, 30]), the changed system variants need to be manually identified and specified by the software engineer, which is a difficult task when a large number of variants need to be specified. Furthermore, a number of approaches have been introduced to realize the system’s design changes to a running system instance (e.g. [31-34]). But, using such approaches is tedious and error prone, where the changes need to be specified at a lower level of abstraction. Thus, in this chapter, we introduce a scenario-driven technique to facilitate the runtime evolution of a context-aware adaptive system. The technique assists the software engineer in performing four tasks as shown in Figure 6-1. First, it enables the software engineer to modify the system’s requirements specification (captured as scenarios) to specify changes to the system requirements. Second, it automatically validates the system scenarios to ensure that the changed adaptation scenarios are still consistent, and the application of them to the modified functional scenarios leads to valid variant specifications of the system.

![Figure 6-1: Tasks for evolving a running context-aware adaptive software system](image-url)
Third, to synthesize changes to the system’s design model from the changed requirements, we introduce a set of change (evolution) patterns. The patterns specify changes to the system’s design model in response to unanticipated changes in the system requirements (scenarios), i.e., changes to the system’s environment, functionality, and adaptive behaviour. Thus, the change patterns can be used for the automatic synthesis of the changes to the system’s design from its requirements changes. In addition, in the case where the software engineer prefers to modify the system’s design model directly, the change patterns provide a guideline that assists the engineer in modifying the system’s design model to incorporate the required changes.

Fourth, the changes to the system’s design model are automatically realized (applied) to the running system by computing the differences between the running system model and its evolved model, and generating adaptation actions to apply the differences to the running system. These actions (e.g. add, remove, or modify an element) are engineered into the executable artifacts of the system during their development time to enable the system’s runtime evolution.

This chapter is organized as follows. Section 6.1 discusses the specification and validation of changes to a system’s requirements (scenarios). In Section 6.2, we present evolution patterns that specify changes to a system’s design model in response to changes in the system scenarios, and how to synthesize the changes to the system design from the scenarios changes. Section 6.3 describes how changes of a system’s design model are realized to the running system.

6.1 Specifying and Validating the System Requirements’ Changes

During the runtime execution of a context-aware adaptive system, the system requirements may evolve by adding functional and/or adaptation requirements or by modifying/removing existing requirements [8, 194]. In this section, we introduce a technique to assist the software engineer in specifying such changes to the system requirements. We also introduce a technique to ensure the consistency and the validity of the changed requirements.

6.1.1 Specifying the Requirements Changes

In Chapter 4, we introduced a technique to capture the requirements of a context-aware adaptive system as two sets of scenarios: functional and adaptation. In addition, the system properties that need to hold while the system is in operation are captured in a form similar to the scenarios.

To capture the changes to a system’s requirements either the specified system scenarios need to be changed or new scenarios need to be introduced. Adding new scenarios or modifying the existing ones depend on the type of the requirements changes. The requirements changes can be
to the system’s functional requirements, or to its adaptation requirements. In the following, we describe these two types of changes in detail.

6.1.1.1 Changes to the Functional Requirements

In response to changes in the system’s functional requirements, new functional scenarios are introduced or the existing functional scenarios are changed. In addition, the system’s properties may change where new properties are introduced for the new scenarios or the existing properties are modified when the functional scenarios that the properties are based on are changed.

(1.1) Adding New Functional Scenarios: A new functional scenario is added to the system to incorporate new functionality, or to take new context information into account.

First, to add a new functionality (feature) to the system, at least a functional scenario needs to be introduced to specify how this feature can be used. Also, the added scenario(s) may introduce new functional and contextual participants, a set of interactions between the participants, and system properties. For example, the travel guide provider may want to enhance the system by adding a restaurant locator service. To accommodate this change, a functional scenario is introduced (i.e. “FS4” in Figure 6-2.A). This scenario starts by a user request to find restaurants that are nearby a location provided by the user. When the request is received by the restaurant locator service, it acquires the user’s food preferences and provides restaurants that match the user preferences. Finally, the user can view the restaurants information and select a suitable restaurant, so that the trip route can be re-planned to take the restaurant location into account.

Figure 6-2: Example changes to the travel guide system’s functional scenarios to add a new feature

The addition of the functional scenario “FS4” introduces a new functional participant (i.e. RestaurantLocator), and its functional and contextual messages with the existing functional and contextual participants (e.g. “FM1” and “CM1”). In addition, a new property that needs to hold
at runtime for the scenario “FS4” is specified (see Figure 6-2.B). The property specifies that the “ProvideRestaurants” interaction should be executed in response to the request of the operation “FindRestaurant”.

Second, taking into account new context information by the system may lead to introducing a functional scenario that describes the use of this information by the system functionality. For example, the system provider may want to consider the vehicle speed and alert the driver when the vehicle speed exceeds the speed limit. To take this information into account, a new scenario is introduced as shown in Figure 6-3. The scenario starts with requesting the speed limit and the vehicle speed. Then, if the vehicle speed is less than the speed limit, the driver is only updated by the vehicle speed. When the vehicle speed exceeds the speed limit, an alert is activated to notify the driver to reduce the speed. Finally, as long as the travel guide system is operating, the scenario “Notify Vehicle Speed” will be repeated. In this scenario, a new contextual participant is added (i.e. the “VehicleInfo”), and the interactions between the user and the route planner and between the route planner and the “TrafficInfo” participants are changed by introducing new functional and contextual messages respectively (see Figure 6-3).

![Figure 6-3: Adding a functional scenario to take new context information into account](image)

(1.2) Changing the Existing Functional Scenarios: The existing functional requirements may change to consider new context information, to take into account changes in the functional and context qualities, or to consider new functionality. First, to take new context information into account, a functional scenario is changed to specify the functional use of the new information. For example, to consider the driving preferences in planning a route, the “plan a route” scenario (shown in Figure 4-6) is changed to include and use such information as shown in Figure 6-4. In this scenario, the contextual message “CM3” is introduced between the “UserInfo” and the route
planner participants, so that route planner can request (acquire) the user’s driving preferences and use it in planning a route.

Second, the system provider may want to improve the functional and context qualities of the system to compete with providers of the same system in the market. To cope with such changes, the functional scenarios need to be changed. For example, the scenario “plan a route” is changed to specify that the traffic information needs to up to the last minute (instead of up to the last two minutes as shown in Figure 4-6), and the “ProvideRoutes1” operation should be executed in less than 5 seconds (instead of 7 seconds as presented in Figure 4-6) as shown in Figure 6-4.

Third, a functional scenario may change to consider a new functionality added to the system. For example, the system provider may want to keep updating the calculated routes during the journey to take the traffic information changes into account. To support that, a new scenario is introduced to perform the routes update task. Then, the existing scenario “FS3” is changed by having a reference to this new scenario as shown at the end of Figure 6-4.
6.1.1.2 Changes to the Adaptation Requirements

To cope with changes in the adaptation requirements of the system, new adaptation scenarios need be introduced or the existing adaptation scenarios need to be modified.

(2.1) Adding New Adaptation Scenarios: Introducing new functional scenarios to the system triggers the need for adding new adaptation scenarios, so that the system adapts properly after incorporating the new functional scenarios. An example adaptation scenario that is added to the travel guide system is “AS3” shown in Figure 6-5. This scenario specifies the adaptations to be performed on the functional scenario “FS5” (shown in Figure 6-3) in response to changes in the speed limit availability. When the speed limit is unavailable, the contextual lifeline “CL1”, the contextual message “CM1”, the functional message “FM2”, and the combined fragment “AT1” are removed from the scenario “FS5” where they depend on the speed limit availability. On the other hand, when the speed limit is available, the above elements are added to the functional scenario “FS5” as shown at the end of Figure 6-5.

![Figure 6-5: An example adaptation scenario that is added to the travel guide system](image)

(2.2) Changing the Adaptation Scenarios: Similar to introducing new adaptation scenarios in response to adding new functional scenarios, the adaptation scenarios need to be modified when the functional scenarios are changed. For example, to allow the user interactions with the restaurant locator service, the system’s main scenario is changed by adding a reference to the
“Find a Restaurant” scenario. A reference to the scenario “Notify Vehicle Speed” is also added, so that the user is alerted when the vehicle speed exceeds the speed limit (see Figure 6-6.A). In response to these changes in the scenario “FS1”, the adaptation scenario “AS1” is modified (as shown in Figure 6-6.B) by adding the adaptation fragment “AD3” that makes the user able to include or exclude the restaurant locator service when needed.

![Figure 6-6: An example change to an adaptation scenario in response to a functional scenario change](image)

6.1.2 Validating the Requirements Changes

The changes to a system’s scenarios may be limited to a small portion of the scenarios, and then validating the whole set of scenarios may be not needed, in particular as it is a time consuming task. To validate the system scenarios while considering the validation that has been performed to the initial set of scenarios (see Chapter 4), in this section, we introduce a technique to validate the system’s scenarios incrementally (i.e. only the changes to the scenarios are validated).

In order to only validate the system scenarios’ changes, we modified the process introduced in Chapter 4 (Section 4.2) for validating the system scenarios. In the modified process (shown in Figure 6-7), tasks for generating the adaptation scripts and the system variants are changed to only generate new adaptation scripts to be checked for consistency, and new system variants to be validated against the relevant system properties.
First, to reduce the number of adaptation scripts to be checked for consistency, we changed the task that is responsible for generating the adaptation scripts to only generate the new scripts as shown in Listing 6-1. To perform this task, the system’s adaptation scenarios are first parsed to identify adaptation conditions that are used for generating valid conditions combinations (see Figure 6-7). Then, for each valid condition combination, an adaptation script is computed (see Lines 3-8 in Listing 6-1). After that, the old consistent scripts are parsed to identify an existing script that is same as the new script. If the new script is not found in the set of consistent scripts, it is added to the list of the new scripts to be checked (see Line 9-17 in Listing 6-1). The result
of this task is the new (or modified) scripts. These scripts are then fed to the script checker to ensure their consistency as shown in Figure 6-7 and described in Section 4.2.1.

Listing 6-1: An algorithm for generating the new adaptation scripts

```java
Void GenerateScripts (AdaptationScenarios AS, ConsistentScripts AdaptationScripts)
1: List NewAdaptationScripts = new List();
2: FOR each Combination C in ConditionsCombinations
3:    AdaptationScript newScript = new AdaptationScript();
4:    List ExecutableFragments = new List();
5:    ExecutableFragments = AS.getExecutableAdaptations();
6:    FOR each Fragment EF in ExecutableFragments
7:        newScript.addActions(EF.getActions());
8:    END FOR
9:    Boolean exist = false;
10:   FOR each Script oldScript in AdaptationScripts
11:      IF oldScript is same as newScript THEN
12:         exist = true;
13:      END IF
14:   END FOR
15:   IF exist == false THEN
16:      NewAdaptationScripts.add(newScript);
17:   END IF
18: END FOR
```

Second, to reduce the time required for validating the system variants (which can be a high number of variants in a large scale system), we consider the system variants that are validated at the development time in generating the system variant specifications that need to be validated when the requirements are changed as shown in Listing 6-2.

Listing 6-2: An algorithm for identifying and generating the new system variants

```java
Void GenerateVariants (FunctionalScenarios FS, ValidVariants Variants, Properties P)
1: List SystemVariants = new List();
2: FOR each Script ST in AdaptationScripts
3:    ST.orderAdaptationActions();
4:    Variant V = applyScript(FS, ST);
5:    SystemVariants.add(V);
6: END FOR
7: FOR each Variant oldVariant in Variants
8:    Properties p1 = IdentifyPropertiesToBeChecked(oldVariant, P);
9:    Properties p2 = oldVariant.getPropertiesToBeChecked(P);
10:   IF p1 is not equals to p2 THEN
11:      V.setPropertiesToBeChecked (difference(p1, p2));
12:     SystemVariants.add(V);
13: END IF
14: END FOR
```

To generate the system variants to be validated, the new adaptation scripts are applied to the system’s functional scenarios to generate the new system variants. These variants are then added to the system variants to be validated (see Lines 2-6 in Listing 6-2). In addition, some of the old variants (validated at the development time) may need to be re-validated because there are new system’s properties need to be checked against them. To identify such variants, the new system properties are parsed to define properties to be checked against each old variant (Lines 8 in
Listing 6-2). Then, the properties that are already checked against the old variant are acquired, and the differences between the already checked properties and the new properties are computed to identify the unchecked properties. In the case where there are unchecked properties, the old system variant is added to the system variants to be validated (Lines 9-13 in Listing 6-2). The result of this process is only the system variant specifications that are affected by the changes to the system’s scenarios (requirements). After identifying the system variants to be validated, the variants are translated to Petri Nets [39], and the relevant system properties are transformed to CTL formulas [40] before being fed to the Romeo model checker [41] for the conformance check as discussed in Section 4.2.2.

6.2 Synthesizing the Changes to the System’s Design Model

In the previous section, we introduced a technique to specify and validate changes to a system’s scenarios. To synthesize changes to the system’s design model from the changed scenarios, the technique presented in Section 5.3 can be used, where a new design model for the system is derived. However, the synthesis process will take a long time for a large scale system as all the system elements need to be generated even in the case where there is only a small change to the system scenarios. In addition, the scenarios’ changes may be performed several times, and then generating the whole system’s design model each time is an inefficient process, especially for large scale systems. As such, there is a need for an efficient technique that only synthesizes the changes to the system’s design model that are corresponding to the scenarios’ changes.

The changes to the system’s functional and adaptation scenarios are performed in response to changes in the system functionality and adaptive behaviour respectively. In addition, the system environment changes trigger the need for modifying the functional (i.e. a functional use of the context information) and adaptation (i.e. making the system able to adapt itself in response to new context changes) scenarios. Based on these categories of unanticipated changes that trigger changes to the system scenarios, we introduce a set of evolution patterns. These patterns specify changes to the system’s design model that are corresponding to changes in the system scenarios. Therefore, they can be used for synthesizing an updated model for the system design from its changed scenarios automatically. In addition, in the case where the software engineer prefers to manually change the system design model in response to the unanticipated changes, the patterns provide a guideline that assists the engineer in directly modifying the system’s design model to incorporate the required changes.

In the following, we describe the evolution (change) patterns, and how to use these patterns to synthesize the updated (evolved) design model of the system from its changed scenarios.
6.2.1 The Evolution Patterns

In general, the system evolves in response to changes in its environment (context), functionality, or adaptive behaviour (as discussed above). As such, we classify the evolution (change) patterns into three groups that are corresponding to the three types of changes. The first group contains patterns that can be used to evolve a system’s design model to cope with unanticipated changes in its *environment* (e.g. taking new context information into account). In the second group, we have a set of change patterns that specify changes to a system’s design in response to changes in its *functionality* (e.g. change an existing system’s functionality). Finally, we have two patterns in the third group to cope with changes to a system’s *adaptive behaviour*. A summary of these evolution patterns is presented in Table 6-1, and we describe the patterns in detail below.

<table>
<thead>
<tr>
<th>Pattern Category</th>
<th>Pattern Name</th>
<th>Pattern Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Changes to a System’s Context</td>
<td>P#1: New Context Attribute</td>
<td>This pattern specifies changes to a system, so that it becomes aware of a new context attribute.</td>
</tr>
<tr>
<td></td>
<td>P#2: New Context Entity</td>
<td>To make a system able to get information about a new context entity, this pattern can be used.</td>
</tr>
<tr>
<td></td>
<td>P#3: Changes to Context Quality</td>
<td>This pattern specifies a system’s design changes to cope with changes in its context attributes’ qualities.</td>
</tr>
<tr>
<td></td>
<td>P#4: Removing Context Entity or Attribute</td>
<td>The context information may become unwanted. Thus, this pattern specifies the changes to the system’s design in response to removing a context attribute or entity.</td>
</tr>
<tr>
<td>2. Changes to a System’s Functionality</td>
<td>P#5: New Functionality</td>
<td>To incorporate new functionality to a system, changes need to be performed into its structure and behaviour. Thus, this pattern specifies the required changes to the system’s model to include such new functionality.</td>
</tr>
<tr>
<td></td>
<td>P#6: Changes to Existing Functionality</td>
<td>Not only a system needs to add new functions, but also its existing ones need to be changed. Thus, this pattern defines possible changes to the existing functionality.</td>
</tr>
<tr>
<td></td>
<td>P#7: Changes to Functional Quality</td>
<td>This pattern specifies changes to a system’s design to cope with changes in its non-functional requirements.</td>
</tr>
<tr>
<td>3. Changes to a System’s Adaptive Behaviour</td>
<td>P#8: New Adaptive Behaviour</td>
<td>To include a new adaptive behaviour in a system, this pattern specifies required changes to the system design.</td>
</tr>
<tr>
<td></td>
<td>P#9: Changes to an Existing Adaptive Behaviour</td>
<td>This pattern specifies changes to the system’s existing adaptive behaviour in response to changes in its context and/or functionality models.</td>
</tr>
</tbody>
</table>

Similar to existing *design patterns* (e.g. [153, 195]), our evolution patterns provide solutions (in terms of changes to a system’s design model) to a number of problems (i.e. the unanticipated changes to the system). But, different from the existing patterns, we identified the patterns from
the different types of the unanticipated changes rather than generalizing solutions to common problems where, to the best of our knowledge, such solutions do not exist yet. Also, the patterns specify changes to the system design instead of supporting the task of designing the system.

6.2.1.1 A Template for the Evolution (Change) Patterns

The evolution patterns specify changes to a system design in response to unanticipated changes in its requirements (scenarios). To document these patterns, we introduce a pattern template. An example evolution pattern following this template is the “add new functionality” pattern shown in Listing 6-3. The pattern consists of a name, a classification, a motivation, scenarios’ changes, system’s design changes, consequences, constraints, and related patterns. First, the pattern name is a unique handle to easily identify the pattern (e.g. new functionality). We also give each pattern an identifier (e.g. P#5) so that the pattern can have references to other patterns (see the last item in Listing 6-3). Second, to facilitate the patterns’ organization, each pattern has a classification (e.g. changes to a system’s functionality). Third, the motivation of the pattern is used to specify when it can be used (i.e. the situation that triggers the need for that pattern). For example, the motivation of the pattern described in Listing 6-3 is that a system provider needs to incorporate new functionality. Fourth, the scenarios’ changes specify the modifications to be performed on the functional and adaptation scenarios of a system to cope with the unanticipated changes. For example, to include a new functionality, a new functional scenario(s) need to be added to define how the new functionality is related to the system’s existing elements as shown in Listing 6-3.

**Listing 6-3:** An evolution pattern to add new functionality to a system’s design model

<table>
<thead>
<tr>
<th>Pattern Id:</th>
<th>P#5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pattern Name:</strong></td>
<td>New functionality.</td>
</tr>
<tr>
<td><strong>Classification:</strong></td>
<td>Changes to a system’s functionality.</td>
</tr>
<tr>
<td><strong>Motivation:</strong></td>
<td>To incorporate new functionality (feature) to a system’s design model. This functionality is either requested by the end users or added by the system provider to attract more users.</td>
</tr>
<tr>
<td><strong>Scenarios’ Changes:</strong></td>
<td></td>
</tr>
<tr>
<td><em>Functional Scenarios:</em></td>
<td>A new functional scenario(s) is added to the system to describe the relationship between the new functionality and the system elements, and how it can be used by the user.</td>
</tr>
<tr>
<td><strong>System’s Design Model Changes:</strong></td>
<td></td>
</tr>
<tr>
<td><em>Functional Structure:</em></td>
<td>To add a new functionality, a functional role and its player(s) are added to provide this functionality and one or more functional contracts are added, so that the new functionality is able to interact with the existing system elements.</td>
</tr>
<tr>
<td><em>Functional Behaviour:</em></td>
<td>The system behaviour needs to be changed either by adding new behaviour processes to capture scenarios in which this new functionality can be used, or by modifying existing behaviour processes to take this functionality into account.</td>
</tr>
<tr>
<td><strong>Consequences:</strong></td>
<td>A new feature is added to the system and it is accessible by the end users.</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
<td>The changes to the system’s design model may trigger a change to an existing system role which in turn needs to have a new player. For a safe runtime evolution, a role’s change is only carried out when it is not active (i.e. the role does not involved in any interactions).</td>
</tr>
<tr>
<td><strong>Related Patterns:</strong></td>
<td>P#1, P#2, and P#8</td>
</tr>
</tbody>
</table>
Fifth, the aim of a change pattern is to specify changes to a system’s design in response to unanticipated changes. These changes can be to the system’s functionality, context, or adaptive behaviour. For example, adding new functionality leads to adding new functional roles and their players, contracts, and behaviour processes (see Listing 6-3). Sixth, when a pattern is applied to a system, it will have consequences. For example, applying the pattern described in Listing 6-3 leads to having a new feature added to an existing system. Seventh, the evolution patterns are used to evolve a running system, and then they need to be applied without negatively impacting the system (i.e. the system moves from one state to another safely). Thus, each pattern needs to have constraints that ensure the safe system evolution (see Listing 6-3). Finally, when a pattern is applied, it may trigger the need for other related patterns to be also applied. In Listing 6-3, the addition of new system functionality leads to the possible use of the new context attribute or new context entity patterns (i.e. P#1 and P#2), where the new functionality may require context information to operate effectively. A new adaptive behaviour pattern may also be applied, so that the system adapts properly after adding the new functionality (i.e. P#8).

6.2.1.2 Evolution Patterns to Cope with Unanticipated Context Changes

While a software system in operation, the environment information that is needed by the system functionality or that triggers its runtime adaptation may change (as discussed in Section 6.1.1). The environment changes include adding new context attribute, adding new context entity (i.e. a context role in our terms, where the role can be viewed as a number of context attributes), and changing the existing context entities and the context quality. To take the environment changes into account, we introduce four evolution patterns (see Table 6-1) that specify required changes to the system’s design model in response to changes in the system scenarios. We describe these four patterns in detail below.

New Context Attribute (Pattern #1): To take a new context attribute into account, the change pattern shown in Listing 6-4 can be used. First, at the scenarios level, a new contextual message is added to a functional or an adaptation scenario based on the intended use of this information. For example, the contextual message “CM3” is added to the functional scenario “FS3”, so that the route planner can take the driving preferences into account in calculating and suggesting the routes as shown in Figure 6-4.

Second, in response to adding a contextual message to a system’s scenario, the structure of the system’s functional or its management composite is changed either by modifying an existing contextual contract to include a new context attribute, or by adding a new contextual contract that includes the new context attribute. In addition, a context provider is added or an existing
provider is modified to make the new context attribute available. Furthermore, to make a safe switch between two context providers of a context role, the switching should only be performed when the context role is not in use as shown in Listing 6-4. For example, to consider the driving preferences during the route planning, the contract “CC2” between the user information and the route planner roles is modified, so that the route planner can request the driving preferences and use it in planning a route (see Listing 6-5). In addition, the context provider bound to the user information role (i.e. the “TravelGuideDB”) is changed to provide the user’s driving preferences as shown in Figure 6-8.

**Listing 6-4:** An evolution pattern to add a new context attribute to a software system

<table>
<thead>
<tr>
<th>Pattern Id:</th>
<th>P#1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pattern Name:</strong></td>
<td>New context attribute</td>
</tr>
<tr>
<td><strong>Classification:</strong></td>
<td>Changes to a system’s context.</td>
</tr>
<tr>
<td><strong>Motivation:</strong></td>
<td>To make the system aware of a new context attribute.</td>
</tr>
<tr>
<td><strong>Scenarios’ Changes:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Functional and Adaptation Scenarios:</strong></td>
<td>A new contextual message is added to an existing scenario.</td>
</tr>
<tr>
<td><strong>System’s Design Model Changes:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Functional and Management Structure:</strong></td>
<td>To add a new context attribute, a contextual contract is modified or a new contextual contract that includes the new attribute is added. In addition, the role that is connected with the new/changed contract is modified. Furthermore, a context provider may be added to provide the new context attribute (if the existing context provider that plays the changed role is not able to provide this attribute).</td>
</tr>
<tr>
<td><strong>Consequences:</strong></td>
<td>A new context attribute becomes available and it can be used by the system.</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
<td>In case of a new context provider needs to be introduced, this provider is only bound with its context role when the role is not active (i.e. the role does not involved in any interactions).</td>
</tr>
<tr>
<td><strong>Related Patterns:</strong></td>
<td>P#6 and P#8</td>
</tr>
</tbody>
</table>

![Figure 6-8: The evolved functional composite of the travel guide system](image)

---

8 Some model elements are drawn in bold to differentiate between them and the unchanged elements.
Third, the addition of a context attribute triggers the need to apply either the pattern “P#6” so that the system functionality can take the context attribute into account to operate effectively, or the pattern “P#8” where the context attribute triggers the system’s adaptation and then a number of reactions to changes in this attribute need to be specified.

**Listing 6-5**: The modified contextual contract “CC2”

<table>
<thead>
<tr>
<th>Contextual Contract ID</th>
<th>CC2:UserInfo_RoutePlanner</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parties:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Context Source:</strong></td>
<td>UserInfo;</td>
</tr>
<tr>
<td><strong>Context Consumer:</strong></td>
<td>RoutePlanner;</td>
</tr>
<tr>
<td><strong>Context Attributes:</strong></td>
<td></td>
</tr>
<tr>
<td>a1: SelectedAttractions;</td>
<td></td>
</tr>
<tr>
<td>a2: DrivingPref;</td>
<td>//The added context attribute</td>
</tr>
</tbody>
</table>

**New Context Entity (Pattern #2):** To consider a new context entity in a software system, a new contextual participant is added to a system’s scenario (see Listing 6-6). In addition, two elements that are corresponding to the new participant are added to the system’s design model: a context role and a context provider. The context role represents the context entity to be added and specifies that the system needs this context information. The context provider is bound with the context role to provide the required context with a specific quality. Furthermore, introducing a context entity triggers the need for applying the pattern “P#1” to specify the attributes of this entity. For example, to take the vehicle information into account, a contextual participant that represents this information is added to the functional scenario “FS5” as shown in Figure 6-3. In addition, the “VehicleInfo” role and its provider (i.e. the On-Board-Diagnoses (ODB) system) are added to the functional composite as shown in Figure 6-8. Furthermore, the contract “CC4” is added to specify that the route planner needs to know the vehicle speed (see Listing 6-7) to alert the driver when his vehicle speed exceeds the speed limit.

**Listing 6-6**: An evolution pattern to add a new context entity to a software system

| Pattern Id: | P#2 |
| Pattern Name: | New context entity. |
| Classification: | Changes to a system’s context. |
| Motivation: | To make the system aware of a new context entity. |
| Scenarios’ Changes: | **Functional and Adaptation Scenarios**: A new contextual participant is added to an existing scenario. |
| System’s Model Changes: | **Functional/Management Structure**: To add a new context entity, a context role is introduced to the functional or to the management composite. This role represents the context entity that is responsible for providing the context information. In addition, a context provider is created and bound to this role. |
| Consequences: | A new context entity becomes available and it can be used by the system’s functionality or by its management. |
| Constraints: | N/A. |
| Related Patterns: | P#1 |
Listing 6-7: The new contextual contract “CC4”

<table>
<thead>
<tr>
<th>Contextual Contract ID</th>
<th>Parties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC4: VehicleInfo_RoutePlanner</td>
<td>Context Source: VehicleInfo; \ Context Consumer: RoutePlanner;</td>
</tr>
<tr>
<td>Context Attributes:</td>
<td>a1: VehicleSpeed;</td>
</tr>
</tbody>
</table>

Changes to the Context Quality (Pattern #3): The system provider may want to change the existing quality of a context attribute or he becomes interested in a new quality for that attribute. To capture this change, the scenario that includes this attribute is modified by adding the new context quality or by changing the existing quality. Then, to reflect this change to the system’s design model, the contextual contract that includes such context attribute is changed to add the new context quality or to modify the existing one (see Listing 6-8). Finally, the patterns “P#8” or “P#9” may be applied, where a number of reactions in response to violation of the specified context quality are specified. For example, the scenario “FS3” is changed (see Figure 6-4) to specify that the traffic information should be up to the last minute (instead of up to the last two minutes as presented in Figure 4-6). The contextual contract “CC1” is also changed (see Listing 6-9), so that the route planner can get the traffic information with freshness up to the last minute for effective calculation of the possible routes.

Listing 6-8: An evolution pattern for changing the context quality

<table>
<thead>
<tr>
<th>Pattern Id:</th>
<th>P#3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern Name:</td>
<td>Change a context quality.</td>
</tr>
<tr>
<td>Classification:</td>
<td>Changes to a system’s context.</td>
</tr>
<tr>
<td>Motivation:</td>
<td>The system provider may want to change the defined quality of a context attribute or he becomes interested in a new quality of that context attribute.</td>
</tr>
<tr>
<td>Scenarios’ Changes:</td>
<td>Functional and Adaptation Scenarios: A new context quality is added to an existing scenario or a specified context quality is modified.</td>
</tr>
<tr>
<td>System’s Design Model Changes:</td>
<td>Contextual Contracts: To change a context attribute’s quality, the contextual contract that contains this attribute needs to be changed to reflect this change which in turn is reflected to the context role that is connected to this contract. Furthermore, a context provider may be needed to provide the new context attribute’s quality (if the existing context provider that plays the changed role is not able to provide the required new quality).</td>
</tr>
<tr>
<td>Consequences:</td>
<td>The system becomes able to get the context information with a new quality.</td>
</tr>
<tr>
<td>Constraints:</td>
<td>Same as the constraints described in Listing 6-4 (P#1).</td>
</tr>
<tr>
<td>Related Patterns:</td>
<td>P#8 and P#9</td>
</tr>
</tbody>
</table>

Removing Context Entity or Attribute (Pattern #4): While the system is in operation, some context information may become unwanted or cannot be used (for privacy reasons). Thus, this context information needs to be removed. At the scenarios level, the removal of the context information is performed by removing a contextual message or a contextual participant, while at the design level it can be either removing a context attribute or a context role and its contracts
from a system’s composite (see Listing 6-10). For example, the travel guide system’s provider may become uninterested in the weather information, and then the weather participant and its interactions with other participants are removed from the scenario “FS2” that is shown in Figure 4-5. In addition, the weather role, the weather service, and the contextual contract “CC3” are removed where they are not needed anymore.

**Listing 6-9**: The modified contextual contract “CC1”

<table>
<thead>
<tr>
<th>Contextual Contract ID</th>
<th>CC1: TrafficInfo_RoutePlanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties:</td>
<td>Context Source: TrafficInfo;</td>
</tr>
<tr>
<td></td>
<td>Context Consumer: RoutePlanner;</td>
</tr>
<tr>
<td>Context Attributes:</td>
<td>a1: TrafficInformation;</td>
</tr>
<tr>
<td></td>
<td>a2: SpeedLimit;</td>
</tr>
<tr>
<td>Context Attributes’ Qualities:</td>
<td>q1: [a1, freshness, LessThan, 1 minute] //The changed context quality</td>
</tr>
</tbody>
</table>

**Listing 6-10**: An evolution pattern to remove a context attribute or a context entity

**Pattern Id**: P#4
**Pattern Name**: Removing context entity or attribute.
**Classification**: Changes to a system’s context.
**Motivation**: The system provider wants to remove context information that is not needed.
**Scenarios’ Changes**:
- Functional and Adaptation Scenarios: Remove a contextual message or a contextual participant.
**System’s Design Model Changes**:
- Functional/Management Structure: A context attribute is removed from a contextual contract, or a context role is removed with its player and the contextual contracts that are connected with that role.
**Consequences**: The system’s become unaware of a context attribute/entity.
**Constraints**: The contextual contracts should be inactive when removed from a system’s composite.
**Related Patterns**: N/A

6.2.1.3 Evolution Patterns to Accommodate Changes in a System’s Functionality

At runtime, the system provider may want to change the system functionality (e.g. including a restaurant locator service), or its non-functional requirements (e.g. reduce the response time of the route planner) to attract more users to the system. To cope with these changes, we introduce three evolution patterns (see Table 6-1).

**New Functionality (Pattern #5)**: To add a new functionality to a system, a set of functional scenarios need to be introduced to capture this functionality as shown in Listing 6-3. In response to adding these scenarios, several changes need to be performed on the system’s design model. First, a functional role and its player(s) are added to provide the new functionality. Then, one or more functional contracts are added, so that the new functionality can interact with the existing system elements. Second, new behaviour processes are added or existing ones are modified to allow the user interactions with the additional functionality (see Listing 6-3).
An example functional scenario that is introduced to include the restaurant locator service is the scenario “FS4” shown in Figure 6-2.A. To reflect this change in the system scenarios to the system design, a number of design changes are performed. First, the system structure is changed by adding the restaurant locator role (i.e. “FR5”) and its player. In addition, the contract “FC4” is added, so that the user is able to locate a restaurant as shown in Figure 6-8. Second, a process corresponds to the scenario “FS4” is added to the travel guide system. This process is described in Listing 6-11 and visualized in Figure 6-9. It defines the sequence of tasks that are performed to find a suitable restaurant. It starts with getting the user’s food preferences. Based on the food preferences, nearby restaurants are suggested. Finally, the user can view the restaurants’ details and select a suitable restaurant, so that the trip route can be re-planned to take into account the restaurant location.

**Listing 6-11:** The description of the “P4: FindRestaurant” process

<table>
<thead>
<tr>
<th>Behaviour Process P4: FindRestaurant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start Event:</strong> FindRestaurantRequested;</td>
</tr>
<tr>
<td><strong>End Event:</strong> RestaurantSelected;</td>
</tr>
<tr>
<td><strong>Tasks:</strong></td>
</tr>
<tr>
<td>t1: {FR5.FindRestaurants, Pre (FindRestaurantRequested), Post (LocateRestaurants)};</td>
</tr>
<tr>
<td>t2: {CR2.GetFoodPref, Pre (LocateRestaurants), Post (FoodPrefAvailable)};</td>
</tr>
<tr>
<td>t3: {FR1.ProvideRestaurants, Pre (FoodPrefAvailable), Post (RestaurantsProvided)};</td>
</tr>
<tr>
<td>t4: {FR5.ViewRestaurant, Pre (RestaurantsProvided), Post (RestaurantViewed)};</td>
</tr>
<tr>
<td>t5: {FR5.SelectRestaurant, Pre (RestaurantViewed), Post (RestaurantSelected)};</td>
</tr>
</tbody>
</table>

| Temporal Constraints: |
| c1: {t3 response to t1 globally} |

**Figure 6-9:** A process that shows the tasks to be executed to find a restaurant
Changes to the Existing System Functionality (Pattern #6): While the system is in operation, not only new requirements are introduced but also the existing requirements are changeable. To incorporate changes to the system’s design model in response to changes in the existing system functionality, we propose the change pattern “P#6”. In this pattern, new functional scenarios are introduced or the existing scenarios are modified to capture the system’s functionality changes (see Listing 6-12). Then, in response to changes in the system scenarios, the system’s functional composite is modified by adding new roles and their players, adding contracts, modifying the existing functional roles and contracts, introducing new behaviour processes, and changing the existing behaviour processes.

Listing 6-12: An evolution pattern to cope with changes in the system’s existing functionality

Pattern Id: P#6  
Pattern Name: Change an existing functionality.  
Classification: Changes to a system’s functionality.  
Motivation: To cope with changes in the existing system functionality.  
Scenarios’ Changes:  
Functional Scenarios: Add new functional scenarios or change the existing scenarios.  
System’s Design Model Changes:  
Functional Structure: The functional structure contains elements that capture the system functionality. Thus, changes to the system functionality leads to modifying the functional roles and contracts of the system’s functional composite, where new roles and contracts are introduced or the existing ones are removed or modified.  
Functional Behaviour: The system behaviour needs to be changed by modifying the existing behaviour processes, or by introducing new ones to cope with the changes in the system functionality.
Consequences: The changed functionality becomes accessible by the users.
Constraints: Same as P#5.
Related Patterns: P#1, P#2, and P#8

Listing 6-13: Part of the modified functional contract “FC2”

<table>
<thead>
<tr>
<th>Functional Contract ID</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FC2: User_RoutePlanner</td>
<td></td>
</tr>
</tbody>
</table>

Parties:  
Role A: User;  
Role B: RoutePlanner;

Interaction Clauses:  
\[\text{i1: } \{\text{PlanRoute (String: CurrentLocation, String: Destination), AtoB, void}\}\];  
\[\text{i2: } \{\text{ProvideRoutes1 (String: Routes), BtoA, void}\}\];  
\[\text{i3: } \{\text{ProvideRoutes2 (String: Routes), BtoA, void}\}\];  
\[\text{i4: } \{\text{SelectRoute (Integer: RouteId), AtoB, Boolean}\}\];  
\[\text{i5: } \{\text{NotifyVehicleSpeed (Integer: VehicleSpeed), BtoA, void}\}\];  
\[\text{i6: } \{\text{AlertVehicleSpeed (Integer: VehicleSpeed, Integer: SpeedLimit), BtoA, void}\}\];

Functional Qualities:  
\[\text{q1: } \{\text{ResponseTime, LessThan , 5 seconds}\}\];  
\[\text{q2: } \{\text{ResponseTime, LessThan , 5 seconds}\}\];

Conversion Clauses (Temporal Constraints):  
\[\text{c1: } \{\text{existence i3 after i1}\}\];  
\[\text{c2: } \{\text{i1 precedes i3 globally}\}\];

For example, the functional scenario “FS5” is added to the travel guide system to notify the user when the vehicle speed exceeds the speed limit as a part of the route planner functionality.
In response to this change in the system scenarios, the system’s functional composite is modified by adding two new interactions to the contract “FC2” as shown in Listing 6-13. The interaction “i5” notifies the user by the current vehicle speed, while the interaction “i6” alerts the driver when the vehicle speed exceeds the speed limit.

**Changes to the Functional System Qualities (Pattern #7):** The system provider may want to change a defined non-functional (quality) requirement, or to add a new quality to compete with different providers of the same system in the market. To cope with this change, the functional scenario that includes the changed functional quality is modified to consider such change (see Listing 6-14). In addition, at the system’s design level, first, a functional contract is modified to incorporate the added/changed quality of service. Second, the functional role that is responsible for the function with the changed quality should be also modified to consider the new quality requirements. Third, a functional player is introduced to satisfy the needed quality of service if the existing player(s) is not able to support the new/changed quality of service as presented in Listing 6-14.

An example changed scenario is “FS3” shown in Figure 6-4. In this scenario, the response time of the “ProvideRotues1” function is changed to be “5” seconds instead of “7” seconds (see Figure 4-6). To represent this change in the travel guide system’s design model, the functional contract “FC2” is modified to incorporate this change in the quality requirements (see Listing 6-13). Similarly, the route planner role is changed to reflect the changed quality requirement in its position description, and a new route planner (player) is added (i.e. “RoutePlanner3” shown in Figure 6-8), so that the system can compute the possible routes in less than five seconds when the “ProvideRotues1” interaction is used.

**Listing 6-14:** An evolution pattern to cope with changes in the system’s non-functional requirements

<table>
<thead>
<tr>
<th>Pattern Id:</th>
<th>P#7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern Name:</td>
<td>Changes to functional system qualities.</td>
</tr>
<tr>
<td>Classification:</td>
<td>Changes to a system’s functionality.</td>
</tr>
<tr>
<td>Motivation:</td>
<td>To cope with changes in the system’s non-functional requirements.</td>
</tr>
<tr>
<td>Scenarios’ Changes:</td>
<td></td>
</tr>
<tr>
<td>Functional Scenarios:</td>
<td>A new functional quality is added, or an existing quality is modified in a functional scenario.</td>
</tr>
<tr>
<td>System’s Design Model Changes:</td>
<td></td>
</tr>
<tr>
<td>Functional Contracts:</td>
<td>To change a functional quality, the functional contract that contains this quality needs to be changed to reflect this change which in turn is reflected to the functional role connected to this contract. Furthermore, a new player may be needed to provide the system’s functionality with the new/changed functional system quality (if the existing player(s) that plays the changed role is not able to provide the required functional quality).</td>
</tr>
<tr>
<td>Consequences:</td>
<td>The system’s becomes able to provide its functionality with the new quality.</td>
</tr>
<tr>
<td>Constraints:</td>
<td>Same as P#5.</td>
</tr>
<tr>
<td>Related Patterns:</td>
<td>P#8 and P#9</td>
</tr>
</tbody>
</table>
6.2.1.4 Evolution Patterns to Incorporate Changes to the Adaptive Behaviour

Not only do the system functionality and environment evolve but also its adaptive behaviour evolves. The changes to the adaptive behaviour are either adding new adaptations or modifying existing ones. First, adding a new functionality to the system triggers the need for new adaptive behaviours. For example, adding the restaurant locator service to the travel guide system means an extra cost to the users. As such, the provider wants to make this service optional where a user can add or remove such service when needed. To do so, a set of adaptations need to be specified to include or exclude this service in response to the user request. Second, the adaptive behaviour decides actions to be performed on the system functionality, and this functionality changes (as discussed above). Thus, the adaptive behaviour that is defined on this functionality needs to also be modified, so that the system can adapt itself properly. To support the evolution of the system adaptive behaviour, we introduce the following two change patterns.

*New Adaptive Behaviour (Pattern #8):* To include new adaptive behaviours, first, a set of adaptation scenarios (or fragments) are added to the system scenarios as shown in Listing 6-15. Example adaptations that are added to the travel guide system are in Figure 6-5 and Figure 6-6. The scenario “AS3” in Figure 6-5 represents the system adaptation in response to changes in the speed limit availability, while the adaptation fragment “AD3” is added to the scenario “AS1” to enable the addition and the removal of the restaurant locator service based on the user’s request to include or exclude such service (see Figure 6-6.B).

**Listing 6-15:** An evolution pattern to add a new adaptive behaviour to the system

| Pattern Id: | P#8 |
| Pattern Name: | Adding new adaptive behaviour. |
| Classification: | Changes to a system’s adaptive behaviour. |
| Motivation: | To introduce a new adaptive behaviour to a system. |
| Scenarios’ Changes: | Adaptation Scenarios: New adaptation scenarios or adaptation fragments. |
| System’s Design Model Changes: | Management Player and Contract: To make the system able to have a reaction to new context changes, a set of adaptation rules need to be added to the system’s management player. In addition, to enable the application of the adaptation actions specified in the new adaptation rules, the management contract needs to be modified to include the new management actions to be performed into the system. |
| Consequences: | New adaptive behaviours are introduced to the system, and the system becomes able to take reactions in response to new context changes. |
| Constraints: | After introducing new adaptation rules, the switching between the system’s old and new management (organizer) players should only be performed when the old player is not in use. |
| Related Patterns: | N/A |

Second, a set of adaptation rules corresponding to the changes in the adaptation scenarios are added to the organizer player. In addition, the management contract between the organizer role and the functional system role is modified to include new adaptation actions. For example, the
rule “AddingRestaurantLocator” that corresponds to the fragment “AD3” of “AS1” (see Figure 6-6.B) is added to the organizer player as shown in Listing 6-16. This rule is activated when the user wants to include the restaurant finder service. In response to this change in the user needs, the system is adapted by adding the restaurant locator role and the contracts “FC4” and “CC5”, binding the restaurant locator role with its player, and adding the process “P4”. In addition, a set of adaptation actions are added to the management contract “MC1” such as “add the behaviour process “P4” to the system’s functional composite” (i.e. the action “ma4” in Listing 6-17).

Listing 6-16: Parts of the description for the modified travel guide system’s organizer player

<table>
<thead>
<tr>
<th>Adaptation Rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule TrafficInfoAvailable:</strong></td>
</tr>
<tr>
<td><em>When</em> ValueChanges (TrafficInfoAvailability);</td>
</tr>
<tr>
<td><em>if</em> TrafficInfoAvailability == True;</td>
</tr>
</tbody>
</table>

| **Rule AddingRestaurantLocator:** |
| *When* ValueChanges (RestaurantLocatorSelected); |
| *if* RestaurantLocatorSelected == True; |

Listing 6-17: Part of the modified management contract “MC1”

<table>
<thead>
<tr>
<th>Management Contract ID:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MC1</strong>: SystemOrganizer_SystemFunctionality</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role A: SystemOrganizer;</td>
</tr>
<tr>
<td>Role B: SystemFunctionality;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Management Actions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td><em>ma4</em>: {AddBehaviourProcess (“P4”, “FindaRestaurant”), AtoB, Boolean};</td>
</tr>
<tr>
<td><em>ma5</em>: {Bind(“RoutePlanner”, “RoutePlanner3”), AtoB, Boolean};</td>
</tr>
</tbody>
</table>

Changes to the System’s Adaptive Behaviour (Pattern #9): The system provider may want to change the system adaptations, or he modified the system functionality and then the adaptive behaviour needs to be changed accordingly. A pattern to specify changes to the system’s design model to cope with changes in its adaptation scenarios is shown in Listing 6-18. In this pattern, a set of adaptation scenarios are modified to take into account changes in the system’s adaptive behaviour. Then, the adaptation rules and the management contract of the system design are modified to reflect the changes in the scenarios. For example, the system provider may want to use the player “RoutePlanner3” (see the rule “TrafficInfoAvailable” in Listing 6-16) instead of the player “RoutePlanner1” (as shown in Listing 5-7) when the traffic information is available. The action “Bind (“RoutePlanner”, “RoutePlanner3””) is also added to the management contract “MC1” as described in Listing 6-17 (i.e. the management action “ma5”).
Listing 6-18: An evolution pattern to change the system’s adaptive behaviour

<table>
<thead>
<tr>
<th>Pattern Id:</th>
<th>P#9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern Name:</td>
<td>Change an existing adaptive behaviour.</td>
</tr>
<tr>
<td>Classification:</td>
<td>Changes to a system’s adaptive behaviour.</td>
</tr>
<tr>
<td>Motivation:</td>
<td>To change the adaptive behaviour in response to either the provider wants to change the way that the system adapts, or the system functionality has been changed and the adaptive behaviour needs to be changed accordingly.</td>
</tr>
<tr>
<td>Scenarios’ Changes:</td>
<td>Adaptation Scenarios: Change adaptation fragments/scenarios.</td>
</tr>
<tr>
<td>System’s Design Model Changes:</td>
<td>Management Player and Contract: For the system to change its reactions to anticipated context changes, a set of adaptation actions specified into the adaptation rules need to be modified. In addition, to enable the application of the changed adaptation actions, the management contract is modified to include these actions.</td>
</tr>
<tr>
<td>Consequences:</td>
<td>The system reactions to the anticipated context changes become different.</td>
</tr>
<tr>
<td>Constraints:</td>
<td>Same as P#8</td>
</tr>
<tr>
<td>Related Patterns:</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.2.2 An Algorithm to Synthesize the Evolved Design Model of the System

In the previous section, we introduced a set of evolution patterns that specify the changes to the system’s design model in response to the unanticipated changes in the system scenarios. Based on the correspondences between the scenarios’ changes and the changes to the system’s design model as discussed above and summarised in Table 6-2, we describe below an algorithm to synthesize the changes to the system’s design model from the changed scenarios automatically. The algorithm (as shown in Listing 6-19) selects proper evolution patterns that are then applied to the system’s design model based on the changes in the scenarios.

**Table 6-2:** The mappings between the scenarios’ changes and the changes to the design model

<table>
<thead>
<tr>
<th>Changes to the System Scenarios</th>
<th>Changes to the System’s Design Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#1: New Contextual Message</td>
<td>Add Context Attribute(s) and Context Provider</td>
</tr>
<tr>
<td>P#2: New Contextual Participant</td>
<td>Add Context Role and its Context Providers</td>
</tr>
<tr>
<td>P#3: New/Changed Context Quality</td>
<td>Add (modify) Context Quality and Context Provider</td>
</tr>
<tr>
<td>P#4: 1- Remove Context Entity</td>
<td>1- Remove Context Role, its Provider, and its Contextual Contracts</td>
</tr>
<tr>
<td>2- Remove Attribute</td>
<td>2- Remove Context Attribute</td>
</tr>
<tr>
<td>P#5: New Functional Scenario</td>
<td>Add Functional Role(s), Functional Contract(s), Functional Player(s), and Behaviour Process</td>
</tr>
<tr>
<td>P#6: Change Functional Scenario</td>
<td>Change Functional Contracts and Behaviour Process, and Add Functional Player(s)</td>
</tr>
<tr>
<td>P#7: Change Functional Quality</td>
<td>Change Functional Contract and Change/Add Functional Player</td>
</tr>
<tr>
<td>P#8: Add New Adaptation Scenario/Fragment</td>
<td>Add Adaptation Rule(s)</td>
</tr>
<tr>
<td>P#9: Change Adaptation Fragment</td>
<td>Change Adaptation Rule</td>
</tr>
</tbody>
</table>

152
Listing 6-19: An algorithm to synthesize an updated system design from the changed scenarios

The algorithm starts by identifying a type for a scenario’s change. The change type can be one of the types summarized in Table 6-2. Then, based on the change type, a proper change pattern is selected and applied using the switch statement shown in Listing 6-19. The first four patterns can be applied to the system’s functional or management composite. Therefore, based on the changed scenario’s type (i.e. functional or adaptation), a selected pattern is applied to the system’s functional composite or to its management composite as shown in Lines 3-7 of Listing 6-19. The next three patterns (i.e. patterns 5-7) are applied to the system’s functional composite
as they are only related to the system’s functionality (Lines 30-38 in Listing 6-19). The last two patterns are applied to the system’s management composite because they are only relevant to the system adaptation (Lines 39-44 in Listing 6-19).

The “apply pattern” methods take a system composite and a scenario change as inputs. Then, they identify the change details and map the scenario change to a set of changes on the system’s design (as discussed in the evolution patterns and summarised in Table 6-2). For example, if the scenario change is “Add contextual participant X”. Then, the changes to the system’s design model are “Add context role X”, “Add context provider X_Provider”, and “Bind role X with the provider X_Provider”.

Following this algorithm, for each change performed on a system scenario, a corresponding set of change actions are performed on the system’s design model. As such, the system’s design model is updated incrementally. In addition, as discussed in Chapter 5, some elements of the system’s design model are related to the system’s solution space and cannot be derived from the system scenarios (e.g. multiple players for a functional role). Therefore, the software engineer is responsible for specifying these elements manually.

6.3 Realizing Runtime Changes to the System

In response to unanticipated changes in a system’s requirements and environment, the system’s design model needs to be modified to take into account such unanticipated changes (see above). The design model changes need to be then realized to the system while in operation to increase the customers’ satisfaction (i.e. runtime evolution) [8, 196]. To enable the runtime evolution of a context-aware adaptive system, we compute differences between the running system’s model and its evolved model (Step 1). Then, adaptation actions corresponding to these differences are applied to the running system (Step 2).

Computing the Models Differences (Step 1). To compute the differences between the running system’s model and its evolved model, we parse the two models to identify elements that have been added, removed, or modified. These changes need to be realized into the running system which is an instance of the executable ROAD model. Thus, to compute the models’ differences either the ROAD model of the running system needs to be translated back to our design model, or the evolved design model is transformed to an executable model (i.e. making the two models at the same level of abstraction). When the running system model is transformed to be in the format of our design model and the differences are computed, adaptation actions corresponding to the differences are at our design model level. Thus, they need to be transformed to executable
actions at the ROAD level. To save the step needed for transforming the generated actions to executable ones, we transform the system’s evolved model to an executable ROAD model (the transformation from our design model to the ROAD model is discussed in Section 5.4). Then, we compute the differences between the two models at the ROAD model level, and generate the executable adaptation actions directly from the differences.

An algorithm for computing differences between the running system’s model and its evolved model, and generating adaptation actions corresponding to the differences as an evolution script is shown in Listing 6-20. The algorithm has two tasks. The first task identifies runtime changes to the system’s functional composite, and the second task computes changes to the management composite. The changes to the functional composite can be in its contracts, roles, and behaviour processes as shown in Lines 8-12 of Listing 6-20, while the management composite’s changes are modifications to its contracts, roles, and adaptation rules (Lines 15-19 in Listing 6-20).

**Listing 6-20:** An algorithm (pseudocode) to compute differences between two system models

```plaintext
1: EvolutionScript ComputeModelsDifferences(SystemModel RM, SystemModel EM) //RM: The Running System Model, EM: The Evolved System Model
2:    EvolutionScript ES= new EvolutionScript();
3:    ComputeFunctionalCompositesDifferences(RM, EM, ES);
4:    ComputeManagementCompositesDifferences(RM, EM, ES);
5:    Return ES;
6: END
7: void ComputeFunctionalCompositesDifferences (SystemModel RM, SystemModel EM, EvolutionScript ES)
8:    identifyChangedFunctionalContracts(RM, EM, ES);
9:    identifyChangedContextualContracts(RM, EM, ES);
10:   identifyChangedRoles(RM, EM, ES);
11:   identifyChangedRolesBinding(RM, EM, ES);
12:   identifyChangedBehaviourProcesses(RM, EM, ES);
13:END
14: void ComputeManagementCompositesDifferences (SystemModel RM, SystemModel EM, EvolutionScript ES)
15:    identifyChangedContextualContracts(RM, EM, ES);
16:    identifyChangedManagementContract(RM, EM, ES);
17:    identifyChangedRoles(RM, EM, ES);
18:    identifyChangedRolesBinding(RM, EM, ES);
19:    identifyChangedAdaptationRules(RM, EM, ES);
20: END
```

To identify changes in the system’s functional contracts, a number of tasks are performed as shown in Listing 6-21. First, the functional contracts in the old and new models of the system are acquired (Lines 1-2 in Listing 6-21). After that, iterating over them finds the added contracts (i.e. contracts that do not exist in the running system). If a contract is new, a set of adaptation actions are inserted into the evolution script to add this new contract (see Lines 3-18 in Listing 6-21). These actions are: add the new contract, and add functional interactions to that contract. Second, the contracts in the running system’s model are parsed to find contracts that do not exist anymore in the evolved system model (Lines 20-25 in Listing 6-21). Then, in the case where a contract is removed, an action to remove this contract is added to the evolution script as shown in Line 27 of Listing 6-21.
Listing 6-21: An algorithm (pseudocode) to identify changes in a system’s functional contracts

```java
void identifyChangedFunctionalContracts(SystemModel RM, SystemModel EM, EvolutionScript ES) {
    ArrayList<FunctionalContract> oldContracts = RM.getFunctionalContracts();
    ArrayList<FunctionalContract> newContracts = EM.getFunctionalContracts();

    // Identify added contracts
    for (FunctionalContract fc in newContracts) {
        boolean exist = false;
        for (FunctionalContract fc1 in oldContracts) {
            if (fc.getId() == fc1.getId()) {
                exist = true;
                break;
            }
        }
        if (!exist) {
            ES.addAction("organizer.addNewContract(\"+fc.getId()+\",\"+fc.getName()+\",\"+fc.getDescription()+\",\"+fc.getStatus()+\",\"+fc.getRoleA()+\",\"+fc.getRoleB();
            ArrayList<Interaction> interactions = fc.getInteractions();
            for (Interaction fi in interactions) {
                ES.addAction("organizer.addNewTerm(\"+fi.getId()+\",\"+fi.getName()+\",\"+fi.getReturn()+\",\"+fi.getParameters()+\",\"+fi.getDirection()+\",\"+fi.getContract();
            }
        }
    }

    // Identify removed contracts
    for (FunctionalContract fc in oldContracts) {
        boolean stillExist = true;
        for (FunctionalContract fc1 in newContracts) {
            if (fc.getId() == fc1.getId()) {
                stillExist = false;
                break;
            }
        }
        if (stillExist) {
            ES.addAction("organizer.removeContract(\"+fc.getId());
        }
    }

    // Identify changed contracts
    for (FunctionalContract fc in newContracts) {
        for (FunctionalContract fc1 in oldContracts) {
            if (fc.getId() == fc1.getId()) {
                ArrayList<Interaction> newInteractions = fc.getInteractions();
                ArrayList<Interaction> oldInteractions = fc1.getInteractions();
                for (Interaction nfi in newInteractions) {
                    boolean InteractionExist=false;
                    for (Interaction ofi in oldInteractions) {
                        if (nfi.getId() == ofi.getId()) {
                            InteractionExist=true;
                            break;
                        }
                    }
                    if (!InteractionExist) {
                        ES.addAction("organizer.addNewTerm(\"+nfi.getId()+\",\"+nfi.getName()+\",\"+nfi.getReturn()+\",\"+nfi.getParameters()+\",\"+nfi.getDirection()+\",\"+nfi.getContract();
                    }
                }
            }
        }
    }
}
```

Third, a contract may still exist in the evolved system model. However, its elements may change where a new interaction is added or an existing interaction is removed or modified. To identify changes to a functional contract in the running system model, we first match between this contract and a contract in the evolved system model. If the two contracts are matched, they are parsed to identify new interactions (Lines 34-42 of Listing 6-21). Then, a set of adaptation
actions are added to the evolution script (i.e. adding new interactions to the existing contract as shown in Lines 43-45 in Listing 6-21). In the same manner, removed and modified interactions can be identified by parsing the interactions in the two contracts and finding which interactions do not exist anymore, or interactions that still exist but with modifications (e.g. a new parameter is added to an interaction).

Similar to identifying changes to the system’s functional contracts, changes to other elements in the running system can be identified. For example, to identify changes to a composite’s roles, an algorithm similar to the one in Listing 6-21 can be used, where instead of parsing the system contracts, the system roles are parsed. Then, the roles in the new and old models are compared to identify added, removed, or modified roles. Because of this similarity in the algorithms for identifying changes in the system’s different elements, we omitted their details. We also ensure the consistency of an evolution script by taking into account the actions dependences during the script generation. For example, a contract should be added before introducing interactions to it as shown in Lines 11-16 in Listing 6-21.

Listing 6-22: Part of the script to evolve the functional composite of the travel guide system

```java
//Adding new roles
organizer.addNewRole("CR4", "VehicleInfo", "Providing information about the vehicle");
organizer.addNewRole("FR5", "RestaurantLocator", "The restaurant finder service");

//Adding new contracts
organizer.addNewTerm("FC4-I1", "FindRestaurant", "void", "String#Location", "AtoB", "FC4");
...

//Changing the existing contracts
organizer.addNewTerm("FC2-I5", "AlertVehicleSpeed", "void", "int#VehicleSpeed* int#SpeedLimit", "BtoA", "FC2");

//Adding new facts
organizer.addNewFact("TrafficInfo", "SpeedLimit");
organizer.addNewFact("VehicleInfo", "VehicleSpeed");

//Changing roles' players binding
organizer.changePlayerBinding("CR4","http://localhost:8080/axis2/services/ODBSystem");
organizer.changePlayerBinding("FR5","http://localhost:8080/axis2/services/RestaurantsFinder");
...

//Update defined processes
organizer.addProcess("BP4", "FindRestaurant","FindRestaurantRequested", "ResturantSelected");
organizer.addBehaviorUnit("BP4_BI2_getFoodPref");
organizer.addTaskToBehaviorUnit("BP4_BI2_getFoodPref","CR2.getFoodPref_Task", "LocateRestaurants", "FoodPrefAvailable");
organizer.addBehaviorUnitToProcesDef("BP4","BP4_BI2_getFoodPref");
...
```

An example evolution script generated by the algorithm presented in Listing 6-20 to evolve the travel guide system in response to changes that were not anticipated at the development time is shown in Listing 6-22. This script includes part of the adaptation actions corresponding to the differences between the system’s functional composite presented in Figure 5-4 and its evolved
model in Figure 6-8. It is worth noting that the generated actions are in the form of executable Java code, so that they can be executed at runtime to evolve the running system.

**Realizing the Models Differences** (Step 2). To execute the adaptation actions generated in the above step, we use **Javassist** to create a Java method on-the-fly. This method communicates with the running system to perform the required adaptations by executing adaptation methods that are engineered into the system’s artifacts during their development time (see Section 5.4). The method has a reference to the running system’s organizer (Lines 5-8 in Listing 6-23), and the Java code corresponds to the differences between the system’s initial and evolved models (i.e. the evolution script variable shown in Line 9 of Listing 6-23). To execute this method, we created a Java class with an evolve method that has an empty body at the development time (i.e. OrganizerLinker, see Line 17 in Listing 6-23). At runtime, we modify this method by replacing its body with the new code that has the adaptation actions to be executed (Lines 19-23 in Listing 6-23). Finally, we create an instance of that class, and use this instance to execute the evolve method (see Lines 24-27 in Listing 6-23).

**Listing 6-23:** Creating a Java method to apply an evolution script to the running system

```
1: String className=SysName+"._organizerStub";
2: String evolveMethod = "public void Evolve( ) {" +
3: "\n\ttry{\n\t\tSystem.out.println("Start Executing the Adaptation Actions...\n");\n\t\t" +
4: "\n\t\t+ className + " organizer= new " + className + "(" +
5: "http://localhost:8080/axis2/services/" + SysName.toLowerCase() + "_organizer");\n\t\torganizer._getServiceClient().getOptions().setSoapVersionURI("+
6: "org.apache.axis.soap.SOAP11Constants.SOAP_ENVELOPE_NAMESPACE_URI");\n\t\t" +
7: "\n\t\tevolutionScript+\n\t\n\t\tSystem.out.println("End of Executing the Adaptation Actions ");\n\t\t}\n\t}\n\tcatch(Exception e){\n\t\te.printStackTrace();\n\t}\n};\n28: myClassLoader = null;
29: myClass = null;
30: systemOrganizer = null;
31: }catch(\Exception e){\n32: e.printStackTrace();\n33: }
```
6.4 Summary

In this chapter, we have introduced a novel scenario-driven technique to facilitate the runtime evolution of a context-aware adaptive software system. First, our technique enables the software engineer to specify and validate unanticipated changes to the system requirements.

Second, we have introduced a set of evolution (change) patterns that specify changes to the system’s design model in response to changes in the system requirements (scenarios). As such, these patterns enable the automatic synthesis of an updated (evolved) model of the system from the changed scenarios. In addition, in the case where, the software engineer wants to modify the system’s design model manually, these patterns provide a guideline that assists him in directly incorporating the needed changes to the design model.

Third, the changes to the system’s design model are automatically realized to the running system by computing the differences between the running system model and its evolved model, and generating adaptation actions to apply the differences on the running system.

The contribution of this chapter is threefold: (1) a technique to incrementally validate a system’s scenarios when they are changed in response to unanticipated changes; (2) a number of evolution patterns that specify changes to a system’s design model in response to changes in its requirements (scenarios), and they enable the automatic synthesis of the evolved system model from the changed scenarios; (3) a technique to realize changes of a system’s design model to a running instance of the system automatically.
Chapter 7: Tool Support

To assist the software engineer in the development and runtime evolution of a context-aware adaptive software system using our approach, we have developed a toolset. The toolset supports the engineer in performing the following tasks:

- **Specifying** the system requirements as a set of functional and adaptation scenarios, and capturing the system properties in a form similar to the scenarios.
- **Checking** the consistency of the adaptation scenarios, and ensuring the validity of the system variant specifications that are generated from the system scenarios.
- **Designing** the system based on its requirements, and realizing the system using the ROAD framework.
- **Evolving** the system at runtime in response to unanticipated changes in its requirements and environment.

This chapter is organized as follows. Section 7.1 gives an overview of the set of developed tools and how they interact with each other. In Section 7.2, we describe a tool that supports the specification of a system’s requirements and the requirements’ changes. A tool for validating a system’s requirements and its changed requirements is presented in Section 7.3. Section 7.4 introduces a tool for the system’s design and realization. A tool for evolving a running context-aware adaptive system is discussed in Section 7.5.

### 7.1 An Overview of the Toolset

The toolset that has been developed to support the software engineer in the development and runtime evolution of a context-aware adaptive system is shown in Figure 7-1. It includes a tool for specifying the system requirements and its properties, a tool for validating the requirements specification, a tool for designing and realizing the system based on its requirements, and a tool to enable the system runtime evolution in response to unanticipated changes in its requirements and environment.

**Requirements Specification Tool**: The requirements specification tool (shown in Figure 7-1) assists the software engineer in specifying a system’s requirements and properties. It supports the engineer in specifying the system’s functional requirements as a set of functional scenarios. Then, based on these scenarios, the system’s adaptation requirements and properties are defined. The adaptation requirements are captured as a set of scenarios that specify possible adaptations to the functional scenarios (e.g. add participant, remove message, etc.). The system properties
are defined as temporal constraints on the functional scenarios in a form similar to the scenarios. The tool also supports the evolution of the system requirements by introducing new scenarios or by changing existing ones.

**Requirements Validation Tool**: The requirements validation tool is used for validating the system requirements. It checks the consistency of the adaptation requirements (scenarios), and ensures the validity of the system variants. To check the adaptation scenarios consistency, these scenarios are parsed to detect inconsistencies including adaptation actions’ redundancy, conflict, and incompleteness. The software engineer is then alerted with the errors when identified. To ensure the validity of the system’s variant specifications, the possible variants are generated and the properties to be checked against them are identified. Then, the variants and their properties are transformed to Petri Nets [39] and CTL [40] formulas respectively to enable their formal validation using the Romeo model checker [41] (see Figure 7-1). This tool supports the initial validation of the system scenarios (requirements) and the incremental validation of the scenarios when they are changed in response to unanticipated changes at the system runtime.

**Design and Realization Tool**: The design and realization tool assists the software engineer in designing the system based on its requirements and in generating an executable system model to be deployed. To ease the task of designing the system, the tool enables the automatic synthesis of the functional and management composites of the system from the system’s scenarios. The software engineer can then manipulate the synthesized design model to added elements that are related to the system’s solution space but cannot be synthesized from the scenarios. To realize the system based on its design model, the tool automatically transforms the system’s design model to an executable ROAD model. When this model is deployed to the ROAD runtime environment, an instance of the system is created which can be adapted as required. In the
ROAD framework [47], the Drools rule engine [149] is used for deciding the system adaptation in response to runtime context changes and the Apache Axis2 [197] is used as the deployment platform as shown in Figure 7-1.

**Runtime Evolution Tool:** To enable the system runtime evolution, we introduced the runtime evolution tool (see Figure 7-1). This tool synthesizes an updated model of the system from the changed scenarios. Then, it computes the differences between the running system’s model and its evolved (updated) model, and generates adaptation actions corresponding to the differences. Finally, Javassist [198] is used for creating a Java method on-the-fly to realize the generated actions to the running system, so that the system can accommodate unanticipated changes in its requirements and environment.

### 7.2 System Requirements Specification

The requirements specification tool consists of two main components as shown in Figure 7-2: a requirements editor and a requirements manager. We describe these components below.

![Requirements Specification Tool](image)

**Figure 7-2:** The architecture of the requirements specification tool

**Requirements Editor:** The requirements editor helps the software engineer in specifying a system’s requirements as two sets of scenarios (i.e. functional and adaptation) and its properties in a form similar to the scenarios using the three editors shown on the right of Figure 7-2. These editors enable the addition of new scenarios and properties, and the removal/modification of existing ones. To implement these editors, we have used the Java graphics library\(^\text{10}\) where a Java class is created for each modelling element (e.g. a scenarios participant). This class implements the “Icon”\(^\text{11}\) class to enable the visualization of that element using the “paintIcon” method. We also introduced a Java class (RequirementsCanvas) that extends the “JPanel”\(^\text{12}\) class to create a drawing canvas where the elements of the requirements model can be added and manipulated.

---

\(^{10}\) [http://docs.oracle.com/javase/tutorial/2d/](http://docs.oracle.com/javase/tutorial/2d/)

\(^{11}\) [http://docs.oracle.com/javase/6/docs/api/javax/swing/Icon.html](http://docs.oracle.com/javase/6/docs/api/javax/swing/Icon.html)

\(^{12}\) [http://docs.oracle.com/javase/6/docs/api/javax/swing/JPanel.html](http://docs.oracle.com/javase/6/docs/api/javax/swing/JPanel.html)
To enable the manipulation of the requirements model’s elements, the “RequirementsCanvas” class implements the “MouseListener” and the “MouseMotionListener” classes.

**Requirements Manager:** The requirements manager is responsible for the management of the requirements’ models created by the engineer (e.g. “System1.casm” shown in Figure 7-2). It has the ability to create and save a requirements model, load a saved requirements model, and save the changes of a requirements model. To implement the requirements model manager, we have used the Java input/output library for creating, reading, and changing the requirements’ models (as files). We also used the “Serializable” class to enable the storage and the retrieval of the elements of the requirements models in a form of Java objects.

Screenshots from the requirements specification tool during the specification of a functional scenario, an adaptation scenario, and system properties from the travel guide system are shown in Figure 7-3, Figure 7-4, and Figure 7-5 respectively. The left part of these Figures displays a set of controls that enables the engineer to do a number of actions such as create a new scenario, create a participant, add a participant to a scenario, add an interaction (message) to a scenario, add a combined fragment, etc.

![Screenshot of the requirements specification tool](image)

**Figure 7-3:** An example functional scenario specified using our tool

The requirements specification tool can be used during a system’s development time and its runtime. At the development time, it is used for specifying the initial set of the system scenarios

---

13 [http://docs.oracle.com/javase/6/docs/api/java/awt/event/package-summary.html](http://docs.oracle.com/javase/6/docs/api/java/awt/event/package-summary.html)
14 [http://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html](http://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html)
15 [http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html](http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html)
and properties. These scenarios and properties can be later modified as needed while the system is still under development. At runtime, in response to the unanticipated changes in the system’s requirements and environment, the system’s scenarios and its properties need to be modified. To accommodate these changes, the tool enables the software engineer to load the requirements model of the deployed software system. Then, he can add new scenarios/properties, and remove or modify existing scenarios/properties.

**Figure 7-4:** An example adaptation scenario specified in the requirements specification tool

**Figure 7-5:** Example system properties specified in a form similar to the scenarios
7.3 Requirements Validation

Two aspects of a system’s requirements need to be checked: adaptation scenarios’ consistency and the validity of the system variants (generated from the functional scenarios). To validate the two aspects, we have developed the requirements validation tool that has two main components (i.e. the consistency checker and the variants validator) as shown in Figure 7-6.

Consistency Checker: The consistency checker enumerates and generates possible adaptation scripts from the adaptation scenarios (maintained in the requirements manager). Then, it checks these scripts to identify inconsistencies in the scenarios, and alerts the software engineer when needed. The output of this component is a set of consistent scripts as shown in Figure 7-6. This component is a Java implementation of the first part of the algorithm presented in Section 4.2. An example use of the consistency checker is shown in Listing 7-1, where adaptation conditions are enumerated (Line 2), adaptation scripts are generated (Lines 3-6), and the generated scripts are checked to detect the inconsistencies in them (Lines 7-8).

![Requirements Validation Tool](image)

**Figure 7-6: The architecture of the requirements validation tool**

**Listing 7-1: Output statements during the execution of the algorithm to generate the system variants**

```plaintext
1: Generating the adaptation scripts ...
2: Number of Conditions is '2' that generate '4' combinations and '2' of them are invalid, and then only '2' scripts are generated.
3: Script#1: AddParticipant|FS3_TrafficInfo AddInteraction|FS3_CM1
4: Script#2: RemoveParticipant|FS3_TrafficInfo RemoveInteraction|FS3_CM1
5: Validating the generated Scripts ...
6: The generated scripts are valid.

7: Applying the Script #1 ...
8: Context Situation: [TrafficInfoAvailabilityEqualTrue] is true, [TrafficInfoAvailabilityNotEqualTrue] is false
9: Actions: #1: FS3_AddParticipant_TrafficInfo #2: FS3_AddInteraction_CM1
10: The Script#1 is applied to 'FS3' and the result is 'FS3V1'
11: Applying the Script #2 ...
12: Context Situation: [TrafficInfoAvailabilityEqualTrue] is false, [TrafficInfoAvailabilityNotEqualTrue] is true
13: Actions: #1: FS3_RemoveParticipant_TrafficInfo #2: FS3_RemoveInteraction_CM1
14: The Script#2 is applied to 'FS3' and the result is 'FS3V2'
15: The variants and the properties that need to be checked against them
16: 'FS3V1' Properties: Existence(FS3_FM3_ProvideRoutes1) | After(FS3_FM1_PlanRoute)
17: 'FS3V2' Properties: Existence(FS3_FM4_ProvideRoutes2) | After(FS3_FM1_PlanRoute)
```
**Variants Validator:** The variants validator generates and validates the system variants. To generate the system variants, the consistent scripts (computed by the consistency checker) are applied to the system’s functional scenarios as described in Section 4.2. Example variants that are generated by this component are shown in Listing 7-1, where two scripts are applied to the travel guide system’s scenarios (Lines 9-18). In addition, the properties to be checked against these two variants are identified (Lines 19-21).

To validate the system variants against their relevant properties, we implemented (in Java) the technique introduced by Bernardi et al. [177] to transform the variant scenarios to Petri nets, and the approach proposed by Dwyer et al. [56] for translating the properties to CTL formulas. This transformation process generates Petri nets and CTL formulas in a format that is acceptable by the Romeo tool (i.e. the XML and CTL files shown in Figure 7-6). Example variants that are generated by our tool are visualized in Figure 7-7 by the Romeo tool. The software engineer can validate these variants visually by playing the token games with the Petri nets [199]. We also created a Java class to enable the communication between our tool and the Romeo tool. This class uses the Runtime class to perform the formal validation by calling the Romeo checker. It also acquires the validation results which are then displayed to the engineer. A screenshot from our tool after checking a system variant against a system property is shown in Figure 7-8.

![Example variants that are generated by our tool and compatible with the Remo tool](http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html)

16 http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html
To validate the system’s requirements when they are changed to incorporate unanticipated changes, the tool is able to identify the adaptation scripts and the system variants that need to be validated as discussed in Section 6.1.2. Thus, only the changed requirements are validated. This feature is supported through the “re-validation” tab shown in Figure 7-8.

7.4 System Design and Realization

To assist the software engineer in designing and realizing context-aware adaptive systems, we introduced the design and realization tool. This tool has four main components: design editor, design models manager, design synthesizer, and model transformer as shown in Figure 7-9.

**Figure 7-9:** The architecture of the design and realization tool

*Design Editor:* The design editor assists the software engineer in designing a context-aware adaptive software system. It has two graphical editors: *system composites* editor and *behaviour process* editor as shown in Figure 7-9. To implement these editors, we followed the technique that is used for implementing the requirements editor (discussed above).
A screenshot of the composites editor is shown in Figure 7-10. The editor has a toolbox as shown on the left of Figure 7-10. This toolbox contains elements that can be used in designing the functional and management composites of a system (e.g. roles, contracts, etc.). The software engineer can use the toolbox to add elements to a system’s design model as shown in the middle of Figure 7-10. Then, using the modelling canvas, the engineer can move the model elements to adjust their positions and to define their connections. The editor also supports the engineer in specifying the detailed description of a system element (if needed). For example, the engineer is able to specify the details of a functional contract as shown in the right of Figure 7-10.

**Figure 7-10:** Designing a system’s functional and management composites

A screenshot from the behaviour process editor during the specification of a process in the travel guide system is shown in Figure 7-11. The editor enables the software engineer to specify a process’s start and end events, and the process’s tasks with their pre and post conditions (see the left part of Figure 7-11). It also supports the visualization of a process as an event-process-chain (EPC) process (see the right part of Figure 7-11).

*Design Models Manager:* The design models manager is responsible for the management of the design models created by the software engineer. It has the ability to create a design model, save a design model, load a saved design model (e.g. “System1.caasdm” shown in Figure 7-9), and save changes to an existing design model. To implement the models manager, we followed the approach of implementing the requirements manager (discussed above).
Figura 7-11: Specifying a system’s behaviour as event-based processes

Design Synthesizer: To ease the task of designing a system, the design synthesizer supports the synthesis of the system’s design model from its scenarios automatically. This component is a Java implementation of the technique introduced in Section 5.3. A screenshot from our tool during the synthesis of the functional and management composites for the travel guide system from its functional and adaptation scenarios is shown in Figure 7-12.

Figura 7-12: Synthesizing a system’s initial design model from its scenarios

Model Transformer: To realize a system designed in our approach, the model transformer generates an executable system from its design model. It transforms the system’s design model to an executable ROAD model, skeletons for functional players and context providers, rules to
generate process events, and an automated organizer player. This component implements the algorithms introduced in Section 5.4. Example executable artefacts that are generated from a system design are shown in the bottom of Figure 7-13. When these artefacts are deployed to the ROAD runtime environment, an instance of the system is created as presented in Figure 7-14. This instance can be adapted in response to runtime context changes.

Figure 7-13: Generating the executable artefacts of a system from its design model

Figure 7-14: An example deployment of the travel guide system to the ROAD runtime environment
7.5 Runtime System Evolution

To evolve a context-aware adaptive system while it is in operation, we developed the runtime evolution tool. The architecture of the tool consists of three main components: design model updater, evolution script generator, and script executor as shown in Figure 7-15.

![Figure 7-15: The architecture of the runtime evolution tool](image)

*Design Model Updater:* At runtime, a system’s functional and adaptation requirements may change. To specify these changes, the requirements specification tool is used (see Section 7.2). In addition, to ensure the validity of the changed requirements, the requirements validation tool is used as discussed in Section 7.3. To reflect the changes in the system requirements to its design model, we introduced the design model updater component. This component is a Java implementation of the technique described in Section 6.2.2, where the changes in the system’s scenarios are reflected automatically to the system’s design by selecting proper patterns and applying them to the system’s design model to consider the scenarios’ changes. In addition, the tool supports the manual change of the system’s design model, where the software engineer can manipulate the design model to incorporate the unanticipated changes in the case that he prefers to change the design model directly.

*Evolution Script Generator:* To identify adaptation actions that need to be applied into the running system in response to the unanticipated changes, we introduced the script generator. It implements the technique introduced in Section 6.3 for computing the differences between the running system model and its evolved model, and generating adaptation actions corresponding to the differences. The result of this component is an evolution script in a form of an executable Java code (e.g. “Script1.scr” shown in Figure 7-15).

*Script Executor:* To execute an evolution script that is generated by the script generator as discussed above, we introduced the script executor component. This component uses Javassist to generate a Java method on-the-fly that communicates with the running system (deployed to the ROAD environment) to execute the script’s adaptation actions as discussed in Section 6.3.
7.6 Summary

In this chapter, we have introduced a toolset to assist the software engineer in the development and runtime evolution of a context-aware adaptive system. The toolset consists of four tools for specifying the system requirements, validating the system requirements, designing and realizing the system, and evolving the system at runtime.

First, the requirements specification tool supports the software engineer in specifying a system’s requirements as two sets of scenarios and capturing its properties in a form similar to the scenarios. It also enables the system’s requirements evolution by introducing new scenarios and changing existing ones.

Second, the requirements validation tool enables the validation of a system’s requirements. It consists of two main elements: consistency checker and variants validator. The consistency checker parses the system’s adaptation requirements to detect the inconsistencies. The variants validator generates the system variants and checks them against the relevant properties.

Third, the design and realization tool enables the software engineer to manually design a context-aware adaptive system. In addition, to ease the task of designing the system, it supports the automatic synthesis of the system’s design model from its scenarios. Furthermore, to realize the system based on its design model, the tool automatically transforms the design model to an executable ROAD model for the deployment.

Fourth, the runtime evolution tool enables the runtime evolution of a context-aware adaptive system in response to unanticipated changes. It synthesizes an updated model for the system design from its changed scenarios and reflects the changes in the system’s design model to the running system, so that it can accommodate the unanticipated changes.
Part III: Evaluation
Chapter 8: Case Studies

In the previous chapters, we have introduced our approach to assisting the software engineer in the development and runtime evolution of context-aware adaptive software systems. It supports a set of engineering tasks, i.e., specification and validation of the system requirements, design and realization of the system based on its requirements, and runtime evolution of the system in response to unanticipated changes.

In this chapter, we demonstrate the applicability of the approach by using it to develop and evolve two case study systems, and analyse the approach effectiveness in assisting the software engineer to specify, validate, design, and evolve context-aware adaptive software systems. The case studies also provide a guideline to the software engineer in how our approach can be used to develop and evolve such type of systems. For each case study, we use our approach to specify the system requirements, validate the requirements, synthesize the system’s design model from its requirements, and evolve the system in response to unanticipated changes. The case studies are: the travel guide software system presented in Chapter 2 and used throughout the thesis to describe the concepts of our approach, and an electronic-exam management system [200-201] that is larger than the travel guide system (see Table 8-1).

This chapter is organized as follows. Section 8.1 presents a summary of the application of our approach to develop and evolve the travel guide system, where the case study details are scattered throughout the thesis. In Section 8.2, we discuss the use of the approach to specify the requirements for the electronic-exam system, validate the system requirements, synthesize the system’s design model from its requirements, realize the system using the ROAD framework, and evolve the system to cope with unanticipated changes in its environment and requirements. Section 8.3 discusses the lessons learnt from using our approach to conduct the case studies.

8.1 The Travel Guide Software System

We have used our approach to specify, validate, design, and evolve the travel guide system described in Chapter 2. In the previous chapters, we have presented a set of examples from the travel guide system to help the discussion of the approach. Below, we present a summary of this case study based on its details that are scattered throughout the thesis.

(1) Specification of the System Requirements: The requirements of the travel guide system are specified as 3 functional scenarios and 3 adaptation scenarios. First, the functional scenarios are a scenario for the main flow of the travel guide system and two scenarios for capturing the
user interactions with the system to “plan a route” and to “find attractions”. The details of these scenarios are presented in Appendix “A”. Second, each functional scenario has a corresponding adaptation scenario. These adaptation scenarios have 5 adaptation fragments that define runtime changes to the functional scenarios in response to changes in the users’ selected features and the context information availability (see Appendix “A”). The system also has 5 global properties and 10 local properties that need to hold at runtime as described in Appendix “A”.

(2) Validation of the System’s Requirements: Using the algorithm discussed in Section 4.2, 5 adaptation conditions are identified. Consequently, we have 32 \(2^5\) possible combinations of the conditions. The two context conditions specified in Figure 4-9 have a conflict, and then half of the generated combinations are removed leading to only 16 valid combinations. Thus, there are 16 adaptation scripts that can be generated leading to 16 system variants (see Table 8-1).

To demonstrate the strength of our algorithm in identifying inconsistencies in the scenarios, we have added a wrong adaptation fragment. This fragment’s condition is same as the condition of the fragment “AD2” in Figure 4-8 but its action is “RemoveCombinedFragment (FS1, RF2)” that contradicts with the action of the fragment “AD2”. After adding this fragment, the number of valid condition combinations is still 16 (discussed above) leading to 16 scripts. But, there is a contradiction between the actions of the added fragment and the fragment “AD2”, and then half of the scripts are inconsistent as the two fragments cannot be executed at the same time. These errors are reported to the software engineer. The engineer then modifies the adaptation scenarios to remove the wrong fragment. After that, the algorithm is able to generate 16 system variants (see the second column in Table 8-1). After being transformed to formal models, these variants and their properties are fed to the Romeo tool for the formal validation.

(3) Synthesis of the System’s Design Model from its Requirements: Following the technique we have introduced in Section 5.3, the design model of the travel guide system is synthesized from its requirements (specified as two set scenarios). The result of the synthesis process is the functional and management composites shown in Figure 5-4 and Figure 5-8 respectively. First, in the functional composite, a set of elements are synthesized: (1) 4 functional roles are derived from the functional scenarios, and 3 functional contracts are introduced between these roles to capture their interactions (an example contract is “FC2” shown in Listing 5-2); (2) 3 context roles corresponding to the contextual participants in the functional scenarios are generated, and 3 contextual contracts are added to specify context information needed by the functional roles (a generated contract is “CC1” shown in Listing 5-3); (3) skeletons for 3 context providers and 4 functional players are synthesized which need to be completed by the system developers and bound to their corresponding roles; (4) 3 behaviour processes corresponding to the 3 functional
scenarios are generated (an example process that is derived from the scenario “FS2” shown in Figure 4-6 is presented in Figure 5-6).

Second, the management composite (as shown in Figure 5-8) is derived from the adaptation scenarios and it contains the following items: (1) a management role and a management contract that connects this role with the functional system role; (2) 2 context roles corresponding to the contextual participants of the adaptation scenarios, and 2 contextual contracts for connecting these roles with the management role (see Figure 5-8); (3) skeletons for 2 context providers to play the context roles; (4) a management player that uses the 10 adaptation rules corresponding to the 5 adaptation fragments (i.e. two rules per fragment to capture its IF and ELSE parts).

Table 8-1 presents a summary for the number of elements that are derived from the travel guide system’s scenarios. The synthesized model is then completed to add elements that cannot be synthesized from the scenarios such as introducing new functional players (e.g. specifying two functional players for the route planner as shown in Figure 5-4), specifying states transfer in the management contract (see Listing 5-8), etc. Finally, the system’s design model is translated to an executable ROAD model using the technique introduced in Section 5.4. When the ROAD model is deployed to the ROAD runtime environment, an instance of the system is created that can be adapted in response to context changes.

(4) Runtime Evolution of the System: To cope with unanticipated changes, a set of tasks are performed. First, the functional and adaptation scenarios of the travel guide system are evolved to take such changes into account using the evolution patterns introduced in Section 6.2.1. The scenarios’ changes are: (1) a functional scenario is introduced to consider the speed limit and alert the driver when needed; (2) a scenario is added to include the restaurant locator service in the system; (3) a scenario is added to update the calculated routes during the journey to take into account the traffic information changes; (4) the scenario “FS2” is changed to consider the user’s driving preferences in suggesting a set of routes; (5) three adaptation scenarios are introduced to specify adaptations to the new functional scenarios in response to runtime context changes; (6) the scenarios “AS1” and “AS3” are modified by including new adaptation fragments to consider the changes in the functional scenarios “FS1” and “FS3”. The details of the evolved scenarios of the travel guide system are described in Appendix “A”.

Second, after introducing changes to the system’s functional and adaptation scenarios, we have used the technique presented in Section 6.1.2 to validate the changed system’s scenarios (requirements). The adaptation scenarios have 10 adaptation conditions. As such, we have 1024 ($2^{10}$) condition combinations. But, the two conditions shown in Figure 4-9 have a conflict, and
then half of the combinations are removed leading to only 512 valid combinations. The valid 512 combinations lead to 512 scripts that can be generated from the adaptation scenarios. The adaptation scenarios are designed carefully, and then the 512 scripts are consistent and generate 512 system variants (see Table 8-1).

Third, an updated model of the system design is synthesized from the changed scenarios by the algorithm discussed in Section 6.2.2. The new model of the system’s functional composite is shown in Figure 6-8 where new functional roles (e.g. restaurant locator), contextual contracts (e.g. “CC5”), etc. have been introduced. In addition, behaviour processes corresponding to the new and changed scenarios are derived. For example, the behaviour process shown in Figure 6-9 is synthesized from the scenario “FS3” in Figure 6-2.A. In the same manner, the changes to the management composite are synthesized. The third column of Table 8-1 shows a summary of the number of elements in the system’s updated model. The updated model is then completed by the software engineer to add elements that are related to the system solution space. For example, a functional player is introduced to play the route planner role (i.e. “RoutePlanner3”).

Finally, to realize the changes of the system’s design model to a running instance of the system, the technique introduced in Section 6.3 is used. The differences between the model of the system instance and its new model are first computed. Then, adaptation actions to realize the differences are generated as a script (see Listing 6-22), and applied to the running system so that it can deal with the unanticipated changes.

**Table 8-1:** Summary of the two case studies

<table>
<thead>
<tr>
<th>The System’s Scenarios and Design Model Elements</th>
<th>Travel Guide System</th>
<th>E-Exam Management System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial</td>
<td>Evolved</td>
</tr>
<tr>
<td>Functional Scenarios</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Adaptation Scenarios/Fragments</td>
<td>3/6</td>
<td>6/10</td>
</tr>
<tr>
<td>System Properties</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>Condition Combinations</td>
<td>32 $(2^8)$</td>
<td>1024$(2^{10})$</td>
</tr>
<tr>
<td>Valid Combinations</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>Generated Variants</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>FC17’s Functional Roles/Players</td>
<td>4/5</td>
<td>5/6</td>
</tr>
<tr>
<td>FC’s Functional/Contextual Contracts</td>
<td>3/3</td>
<td>4/5</td>
</tr>
<tr>
<td>FC’s Context Roles/Providers</td>
<td>3/4</td>
<td>4/5</td>
</tr>
<tr>
<td>FC’s Behaviour Processes</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>MC18’s Context Roles/Providers</td>
<td>2/2</td>
<td>2/2</td>
</tr>
<tr>
<td>MC’s Contextual Contracts/Attributes</td>
<td>2/4</td>
<td>2/9</td>
</tr>
<tr>
<td>MC’s Adaptation Rules</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

17 The system’s functional composite
18 The system’s management composite
8.2 An Electronic-Exam Management System

An electronic-exam (e-exam, for short) management system is a software system that enables a lecturer to design a computer-based exam, the students to take the exam, and the lecturer (or an assessor) to score the exam [200-202]. The e-exam system can be used by different universities, lecturers, and students. As such, the e-exam system needs to have the ability to be customized to suit different universities’ requirements, to be adapted at runtime based on its users context (e.g. a student is only able to take an exam on the campus), and to evolve to include new features or to take new context information into account.

In this section, we present the details for specifying, validating, designing, and evolving the context-aware adaptive e-exam system. To demonstrate that our approach supports the e-exam system’s evolution, we assume that a set of the system’s functional and adaptation requirements are not known during the initial system development. Then, we describe how to evolve the e-exam system at runtime to incorporate this set of requirements.

8.2.1 Specifying the System Requirements

The e-exam system has a set of functional requirements that specify functions that need to be provided by the system, and a set of adaptation requirements to define variations in the system functionality to suit different needs and contexts.

(1) The Functional Requirements: The system is used by a student or a lecturer, and then we have two main scenarios in the e-exam system as shown in Figure 8-1. First, the scenario “FS1” shown in Figure 8-1.A describes a student’s interactions with the system where the student can request the login operation from the system. After successful login, he can take an exam or view his results. The functional scenario that describes the sequence of interactions between a student and the exams’ manager while he is taking an exam (i.e. “FS3”) is shown in Figure 8-2.A. The scenario starts by the student request to take an exam. Then, a set of questions are provided to the student. To view the questions, the functional scenario “FS4” is used where each question is viewed based on its type as shown in Figure 8-2.B. After that, the student can answer or skip the questions. Finally, he can review his answers and end the exam.

Second, similar to the functional scenario “FS1”, the scenario “FS7” shown in Figure 8-1.B describes a lecturer’s interactions with the e-exam system. This scenario shows that the lecturer is able to create a new account (in the case that he is new lecturer), login to the system, design an exam, preview a designed exam and modify it when need, score an exam performed by a student, and logout from the system.
Figure 8-1: Two scenarios that describe student and lecturer interactions with the e-exam system

Figure 8-2: A student’s interactions with the e-exam system to take an exam

Table 8-2 summarises the system’s functional scenarios, where we give small descriptions for the scenarios and they are presented in detail in Appendix “B”. In general, there are: (1) two main scenarios to enable the interactions between a student and the system (i.e. “FS1”) and
between a lecturer and the system (i.e. “FS7”), (2) two scenarios to enable the login of a student and a lecturer (i.e. “FS2” and “FS8”), (3) three scenarios that describe the interactions between the student and the system while he is taking an exam (i.e. “FS3”, “FS4”, and “FS5”), (4) a scenario that enables a student to view his results (i.e. “FS6”), (5) a scenario to make a lecturer able to create an account (i.e. “FS9”), (6) four scenarios to enable the design of an exam (i.e. “FS10”, “FS11”, “FS12”, and “FS13”), and (7) a scenario to score an exam (i.e. “FS14”).

Table 8-2: The e-exam system’s functional scenarios

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Scenario Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS1: Using the System by a Student</td>
<td>The possible interactions between a student and the e-exam system as shown in Figure 8-1.A.</td>
</tr>
<tr>
<td>FS2: Student Login</td>
<td>To allow a student’s interactions with the system, he needs to login by following the sequence of interactions in this scenario.</td>
</tr>
<tr>
<td>FS3: Take an Exam</td>
<td>The set of tasks that need to be performed by a student to take an exam (see Figure 8-2.A).</td>
</tr>
<tr>
<td>FS4: View a Question</td>
<td>This scenario is used to display a question to a student. The question is displayed based on its type (multiple choices, matching, etc.) as shown in Figure 8-2.B.</td>
</tr>
<tr>
<td>FS5: Review Answers</td>
<td>This scenario enables the student to review his answers and modify them when needed.</td>
</tr>
<tr>
<td>FS6: View Results</td>
<td>In this scenario, a student can view an exam(s) result.</td>
</tr>
<tr>
<td>FS7: Using the system by a Lecturer</td>
<td>The scenario describes functional interactions that can be exchanged between a lecturer and the e-exam system (see Figure 8-1.B).</td>
</tr>
<tr>
<td>FS8: Lecturer Login</td>
<td>A set of interactions to be followed by a lecturer to login to the system.</td>
</tr>
<tr>
<td>FS9: Create a Lecturer Account</td>
<td>In the case of new lecturer, this scenario is used to register the lecturer.</td>
</tr>
<tr>
<td>FS10: Design an Exam</td>
<td>A set of tasks to design an exam or to modify an existing exam.</td>
</tr>
<tr>
<td>FS11: Modify Questions</td>
<td>This scenario enables a lecturer to add, remove, or modify a question.</td>
</tr>
<tr>
<td>FS12: Add a Question</td>
<td>The set of interactions between the e-exam system and a lecturer to add a question based on its type.</td>
</tr>
<tr>
<td>FS13: Preview an Exam</td>
<td>To check an exam after it has been designed, this scenario enables the lecturer to view the exam from the students’ perspective.</td>
</tr>
<tr>
<td>FS14: Score an Exam</td>
<td>The set of tasks to evaluate/assess an exam performed by a student.</td>
</tr>
</tbody>
</table>

(2) The Adaptation Requirements: At runtime, the e-exam system needs to be adapted to consider a university’s requirements (i.e. the features they want to be included in the system), a lecturer location (e.g. on the campus), and a student location (e.g. off the campus). We describe below a set of adaptation scenarios that capture the e-exam system’s adaptation requirements.

An example adaptation scenario that specifies adaptations to the scenario “FS1” is “AS1” shown in Figure 8-3. It has an adaptation fragment that removes the sequence reference “take an exam” when a student is not on the campus, while this sequence reference is added otherwise (i.e. the student can only take an exam on the campus).
Another adaptation scenario is “AS3” shown in Figure 8-4. This scenario has two adaptation fragments “AD1” and “AD2”. The fragment “AD1” is used for adapting the functional scenario “FS7” so that a lecturer is only able to score an exam while on the campus (due to some security reasons). This fragment removes the scenario that can be used by the lecturer to score the exam when he is not on the campus while it is added otherwise. The fragment “AD2” is used to adapt the system by adding/removing the feature that enables the registration of a new lecturer (i.e. the scenario “create lecturer account” as shown in Figure 8-1). This feature may be removed when the university has another sub-system for registering the lecturers.

Table 8-3 summaries the different adaptation scenarios in the e-exam software system and their details are presented in Appendix “B”. Different form the travel guide system case study,
not all functional scenarios have their corresponding adaptation scenarios where some scenarios should be executed in the same manner in different context situations. For example, the scenario “FS3” for taking an exam is not affected by the context changes, and then it behaves in the same manner in the different system variants.

Table 8-3: The e-exam system’s adaptation scenarios

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Scenario Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1: Adapting “FS1” (1 adaptation fragment)</td>
<td>Based on the user location, the option to take an exam is enabled or disabled where the student can only take an exam on the campus (see Figure 8-3).</td>
</tr>
<tr>
<td>AS2: Adapting “FS4” (4 adaptation fragments)</td>
<td>The university may want to restrict types of questions to be taken in an exam, and then the scenario “FS4” needs to be changed to only view the questions that are permitted by the university.</td>
</tr>
<tr>
<td>AS3: Adapting “FS7” (2 adaptation fragments)</td>
<td>Due to some security reasons, a lecturer is only able to score an exam while on the campus. Thus, this scenario changes the possible interactions between the lecturer and the e-exam system based on his location. In addition, the feature to create a lecture account may be removed from the system where it is performed by another registration sub-system (see Figure 8-4).</td>
</tr>
<tr>
<td>AS4: Adapting “FS10” (1 adaptation fragment)</td>
<td>This scenario adapts the functional scenario “FS10: design an exam” when the lecturer preferences are not available.</td>
</tr>
<tr>
<td>AS5: Adapting “FS12” (4 adaptation fragments)</td>
<td>Based on allowed questions’ types, this scenario adapts the functional scenario “adding exam questions”, so that only allowed questions can be added.</td>
</tr>
</tbody>
</table>

(3) The System Properties: While the e-exam system in operation, a set of system properties need to hold. Example properties are shown in Figure 8-5. Figure 8-5.A shows the properties to be preserved during the execution of the functional scenario “FS1”, while the properties to hold while the scenario “FS3” is executing are shown in Figure 8-5.B.

First, the property “PO1” shown in Figure 8-5.A specifies that a student needs to login using the sequence reference “RF1” before viewing an exam result, while the property “PO2” states that a student needs to login before taking an exam. The properties “PO3” and “PO4” in Figure 8-5.A specify that a student should not be able to take an exam when he is off the campus (i.e. “AB1”), while he is able to take the exam on the campus (i.e. “ET1”).

Second, in Figure 8-5.B, the property “PO1” defines that the “StartExam” interaction should precede the “EndExam” interaction. The property “PO2” specifies that the “ProvideQuestion” operation must precede the “ViewQuestion” task. The properties “PO3” and “PO4” specify that in response to a question provided to a student (while he is taking an exam) either he provides an answer (i.e. “RS1”) or he skips that question (i.e. “RS2”).

Table 8-4 shows a summary of the local and global properties that need to hold while the e-exam system in operation (i.e. 26 local properties and 27 global properties). These properties are described in detail in Appendix “B”.

182
Figure 8-5: Example properties that need to be preserved in the e-exam system

Table 8-4: Summary of the e-exam system’s properties

<table>
<thead>
<tr>
<th>The Functional Scenarios</th>
<th>The System Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local Properties</td>
</tr>
<tr>
<td>FS1: Using the System by a Student</td>
<td>3</td>
</tr>
<tr>
<td>FS2: Student Login</td>
<td>0</td>
</tr>
<tr>
<td>FS3: Take an Exam</td>
<td>0</td>
</tr>
<tr>
<td>FS4: View a Question</td>
<td>8</td>
</tr>
<tr>
<td>FS5: Review Answers</td>
<td>0</td>
</tr>
<tr>
<td>FS6: View Results</td>
<td>0</td>
</tr>
<tr>
<td>FS7: Using the System by a Lecturer</td>
<td>5</td>
</tr>
<tr>
<td>FS8: Lecturer Login</td>
<td>0</td>
</tr>
<tr>
<td>FS9: Create a Lecturer Account</td>
<td>0</td>
</tr>
<tr>
<td>FS10: Design an Exam</td>
<td>2</td>
</tr>
<tr>
<td>FS11: Modify a Question</td>
<td>0</td>
</tr>
<tr>
<td>FS12: Add a Question</td>
<td>8</td>
</tr>
<tr>
<td>FS13: Preview an Exam</td>
<td>0</td>
</tr>
<tr>
<td>FS14: Score an Exam</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total: 53 Property</strong></td>
<td><strong>26</strong></td>
</tr>
</tbody>
</table>
8.2.2 Validating the System Requirements

To validate the e-exam system’s requirements, we have used the technique described in Section 4.2. Using this technique, we have checked the consistency of the adaptation scenarios, and the validity of the system variants that are generated from the functional scenarios.

(1) Checking the Adaptation Scenarios Consistency: The algorithm that is proposed to check the consistency of the adaptation scenarios starts with enumerating the adaptation conditions. The e-exam system has 5 adaptation scenarios that have 12 adaptation fragments as presented in Table 8-3. Four of the adaptation conditions are repeated where the scenarios “FS4” and “FS12” are adapted in response to the same context situations (i.e., changes in the allowed questions’ types). Therefore, we have 8 adaptation conditions leading to 256 ($2^8$) condition combinations. These combinations generate 256 scripts. We have specified the adaptation scenarios carefully, and then the 256 scripts are consistent.

(2) Validating the System Variants: After checking the adaptation scripts’ consistency, a set of system variants are generated by applying the scripts to the functional scenarios. Thus, there are 256 variants corresponding to the consistent scripts. The properties that need to be checked against these variants are also identified. Finally, the variants and their properties are translated to Petri nets and CTL formulas, so that the Romeo tool is used for validating the variants.

8.2.3 Synthesizing the Design Model of the E-exam System

Using the algorithms presented in Section 5.3, the functional and management composites of the e-exam system are derived from its scenarios. The number of design elements that are generated from the system scenarios is shown in the fourth column of Table 8-1. Below, we discuss these elements and their correspondences with the scenarios’ elements.

(1) The Functional Composite: In the scenarios, first, we have 5 functional participants and a contextual participant, and then they are translated to their corresponding functional and context roles as shown in Figure 8-6. Second, the functional messages are transformed to functional interactions (specified in functional contracts) between the system’s functional roles. An example generated functional contract is “FC6” shown in Listing 8-2. This contract has a set of interactions that can be exchanged between the student and the exams manager roles. In the same manner, the contextual messages between the exams designer and the lecturer preferences participants are used for deriving the contextual contract “CC1”. Third, for each synthesized role, a skeleton of a player (or a context provider) is created and bound to that role as described in Listing 8-1. Fourth, a set of behaviour processes are generated from the functional scenarios,
where each scenario is translated to a process (see the end of Listing 8-1). An example process is described in Listing 8-3 and is visualized in Figure 8-7. This process is corresponding to the functional scenario “FS1” shown in Figure 8-1.A.

![Figure 8-6: The e-exam system’s functional composite that is synthesized from the scenarios](image)

**Listing 8-1:** A high level description for the functional composite of the e-exam system

---

**Functional Composite E-exam System Functionality**

**Functional Roles:**
- FR1: Student {...};
- FR2: Lecturer {...};
- FR3: AccountsManager {...};
- FR4: ExamsManager {...};
- FR5: ExamsDesigner {...};

**Functional Contracts:**
- FC1: Student_AccountsManager {...};
- FC2: Lecturer_AccountsManager {...};
- FC3: Lecturer_ExamsDesigner {...};
- FC4: Lecturer_ExamsManager {...};
- FC5: ExamsDesigner_ExamsManager {...};
- FC6: Student_ExamsManager {...};

**Functional Players:**
- FP1: StudentGUI1 CanPlay Student;
- FP2: StudentGUI2 CanPlay Student;
- FP3: LecturerGUI1 CanPlay Lecturer;
- FP4: LecturerGUI2 CanPlay Lecturer;
- FP5: Authenticator CanPlay AccountsManager;
- FP6: ExamsManagement CanPlay ExamsManager;
- FP7: ExamsDesignerGUI CanPlay ExamsDesigner;

**Context Roles:**
- CR1: LecturerePref {...};

**Contextual Contracts:**
- CC1: LecturerePref_ExamsDesigner {...};

**Context Providers:**
- CP1: LecturerProfile CanProvide LecturerePref;

**Behaviour Processes:**
- P1: UsingSystemByStudent {...};
- P2: StudentLogin {...};
- ...
- P13: PreviewExam {...};
- P14: ScoreExam {...};
Listing 8-2: Part of the functional contract “FC6”

<table>
<thead>
<tr>
<th>Functional Contract ID</th>
<th>FC6: Student_ExamsManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties:</td>
<td></td>
</tr>
<tr>
<td>Role A: Student;</td>
<td>Role B: ExamsManager;</td>
</tr>
<tr>
<td>Interaction Clauses:</td>
<td></td>
</tr>
<tr>
<td>i1: {StartExam(), AtoB, void};</td>
<td>i2: {EndExam(), AtoB, void};</td>
</tr>
<tr>
<td>i3: {ProvideQuestion(Question), BtoA};</td>
<td>i4: {ProvideAnswer(Answer), AtoB, void};</td>
</tr>
<tr>
<td>Conversion Clauses (Temporal Constraints):</td>
<td></td>
</tr>
<tr>
<td>c1: {i4 response to i3}</td>
<td>c2: {i1 precedes i2 globally}</td>
</tr>
</tbody>
</table>

Listing 8-3: The description of the process “P1: UsingSystemByStudent”

<table>
<thead>
<tr>
<th>Behaviour Process P1: UsingSystemByStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Event: LoginRequested;</td>
</tr>
<tr>
<td>End Event: Loggedout;</td>
</tr>
<tr>
<td>Tasks:</td>
</tr>
<tr>
<td>t1: {FR3.RequestLogin, Pre (LoginRequested), Post (RequestStudentInfo)};</td>
</tr>
<tr>
<td>t2: {StudentLogin, Pre (RequestStudentInfo), Post (LoginSuccess)};</td>
</tr>
<tr>
<td>t3: {TakeExam, Pre (LoginSuccess), Post (ExamTaken)};</td>
</tr>
<tr>
<td>t4: {ViewResult, Pre (LoginSuccess), Post (ResultViewed)};</td>
</tr>
<tr>
<td>t5: {FR3.Logout, Pre (ExamTaken or ResultViewed), Post (Loggedout)};</td>
</tr>
<tr>
<td>Temporal Constraints:</td>
</tr>
<tr>
<td>c1: {t2 precedes t3 globally}</td>
</tr>
<tr>
<td>c2: {t2 precedes t4 globally}</td>
</tr>
</tbody>
</table>

Figure 8-7: A behaviour process corresponding to the functional scenario “FS1”

(2) The Management Composite: The management composite is derived from the adaptation scenarios. This composite structure is shown in Figure 8-8 and is described in Listing 8-4. First, it has a set of roles corresponding to the participants in the adaptation scenarios. The composite has two context roles that represent the system’s selected features and the location of a student or a lecturer. The location information can be identified using the location of the device that is
used for the interactions with the system (e.g. a computer). The composite also has an organizer role to decide the adaptation actions in response to context changes and a role that represents the system functionality. Second, the composite roles interact with each other through contextual and management contracts as described in Listing 8-4. For example, to apply a set of adaptation actions to the e-exam system’s functional composite, a management contract is formed between the organizer and the system functionality roles (i.e. “MC1”). This contract contains the actions to be applied into the system’s functional composite as shown in Listing 8-5.

![Figure 8-8: The structure of the e-exam system’s management composite](image)

**Listing 8-4: A high level description of the management composite for the e-exam system**

<table>
<thead>
<tr>
<th>Management Composite E-exam System Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management Role:</td>
</tr>
<tr>
<td>MR1: SystemOrganizer {…};</td>
</tr>
<tr>
<td>Functional System Role:</td>
</tr>
<tr>
<td>SR1: SystemFunctionality {…};</td>
</tr>
<tr>
<td>Context Roles:</td>
</tr>
<tr>
<td>CR1: Location {…};</td>
</tr>
<tr>
<td>CR2: SystemFeatures {…};</td>
</tr>
<tr>
<td>Contextual Contracts:</td>
</tr>
<tr>
<td>CC1: Location_SystemOrganizer {…};</td>
</tr>
<tr>
<td>CC2: SystemFeatures_SystemOrganizer {…};</td>
</tr>
<tr>
<td>Management Contract:</td>
</tr>
<tr>
<td>MC1: SystemOrganizer_SystemFunctionality {…};</td>
</tr>
<tr>
<td>Players:</td>
</tr>
<tr>
<td>OP1: OrganizerPlayer CanPlay SystemOrganizer;</td>
</tr>
<tr>
<td>SC1: ExamSystemFunctionalComposite CanPlay SystemFunctionality;</td>
</tr>
<tr>
<td>Context Providers:</td>
</tr>
<tr>
<td>CP1: LocationSensor CanProvide Location;</td>
</tr>
<tr>
<td>CP2: SystemDB CanProvide SystemFeatures;</td>
</tr>
</tbody>
</table>

Third, a synthesized management player is shown in Listing 8-6. In general, each fragment in the adaptation scenarios is transformed either to a rule (if the fragment does not have an ELSE part) or to two rules when the fragment has an ELSE part. Example synthesized rules are shown in Listing 8-6. These rules are corresponding to the adaptation scenario “AS1” shown in Figure 8-3. The rule “EnableTakingExam” is derived from the IF-part of the fragment “AD1” while the ELSE-part of this fragment is transformed to the rule “DisableTakingExam”.

187
8.2.4 Evolving the E-exam Management System

During the lifetime of the e-exam system, the system provider may want to add new features (e.g. enable students to take an exam practice), take new context information into account (e.g. a student’s preferences), or change the system adaptation to take the new context information and functionality into account while the system adapts at runtime. To evolve the e-exam system in response to these unanticipated changes, a number of tasks need to be performed as discussed in Chapter 6. In this section, we present the details of performing these engineering tasks to evolve the e-exam system at runtime.

8.2.4.1 Evolving the System Scenarios

To cope with the unanticipated changes in the e-exam system, the functional and adaptation scenarios of the system need to be changed by adding new scenarios or modifying existing ones. The system properties that need to hold at runtime may also need to be changed.
First, the changes to the functional scenarios are introduced to add new functionality and to take new context information into account. Table 8-5 summarises the changes to the functional scenarios in response to the unanticipated changes, where two scenarios are changed and four functional scenarios are introduced (see Appendix “B” for further details).

Table 8-5: The changes to the functional scenarios of the e-exam system

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Scenario Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS1: Using the System by a Student</td>
<td>The possible interactions between a student and the e-exam system are changed to include new interactions as shown in Figure 8-9.A.</td>
</tr>
<tr>
<td>FS7: Using the system by a Lecturer</td>
<td>The scenario “FS7” is changed to describe new interactions that can be performed by a lecturer as shown in Figure 8-9.B.</td>
</tr>
<tr>
<td>FS15: Take a Training</td>
<td>A set of tasks to be followed by a student to take an exam practice (see Figure 8-10).</td>
</tr>
<tr>
<td>FS16: Create a Student Account</td>
<td>This scenario is used for adding a new student to the system.</td>
</tr>
<tr>
<td>FS17: View an Exam’s Results</td>
<td>This scenario enables the lecturer to view a summary of an exam’s results, view a student’s result, and print the results.</td>
</tr>
<tr>
<td>FS18: Select an Exam Type</td>
<td>The e-exam system supports two modes in taking an exam: solo (single student) or group (multiple students solve the exam together). In addition, the exam can be stored in a student computer (offline test) or on a server (online test). As such, this scenario allows the student to select a suitable mode and type for an exam.</td>
</tr>
</tbody>
</table>

**Figure 8-9:** Changes to the e-exam system’s functional scenarios “FS1” and “FS7”

Example changed functional scenarios are “FS1” and “FS7” presented in Figure 8-1. The changes to these scenarios allow students and lecturers to use the new system features such as
create a student account, take training for the exams, and view results of the students in an exam as shown in Figure 8-9.

An example functional scenario that is added to the e-exam system is the scenario “FS15” shown in Figure 8-10. It specifies a student’s interactions with the e-exam system to practice for the exams. The scenario starts with suggesting a number of exams for the student based on his previous attempts. Then, he can select an exam and start it. Finally, he answers the questions in the exam and ends the exam practice.

![Figure 8-10: A scenario that shows the tasks to be performed by a student to take an exam practice](image)

Second, for the e-exam system to adapt properly after introducing changes to the functional scenarios, a set of changes need to be incorporated into the adaptation scenarios. For example, the scenario “AS1” is changed in response to changes in the scenario “FS1” (see Figure 8-11) by adding two adaptation fragments: “AD2” and “AD3”. These fragments are used for adding and removing the features that allow a student to take an exam practice and to create an account.

A summary of the changes to the system’s adaptation scenarios is shown in Table 8-6 and they are described in Appendix “B” in detail. These changes include modifying the scenarios “AS1” and “AS3” to take into account the changes in the functional scenarios “FS1” and “FS7”, and adding two new adaptation scenarios (i.e. “AS6” and “AS7”).
Table 8-6: Incorporating changes to the e-exam system’s adaptation scenarios

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Scenario Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1: Adapting “FS1” (3 fragments)</td>
<td>This scenario is changed by adding two fragments that enable the addition and removal of “create a student account” and “take training” features.</td>
</tr>
<tr>
<td>AS3: Adapting “FS7” (3 fragments)</td>
<td>The feature that is added to e-exam system to enable the lecturer to view an exam’s results is optional. Thus, this scenario is changed to reflect that.</td>
</tr>
<tr>
<td>AS6: Adapting “FS15” (1 fragment)</td>
<td>This scenario includes a fragment to adapt the scenario “FS15” in response to the availability of the user information.</td>
</tr>
<tr>
<td>AS7: Adapting “FS18” (2 adaptation fragments)</td>
<td>There are two modes for taking an exam: solo or group. In addition, the exam can be online or offline. Thus, this scenario enables the university to select from such features.</td>
</tr>
</tbody>
</table>

Third, the system properties are changed to reflect the changes in the functional scenarios, where new properties are introduced. These new properties need to be preserved while the new or the changed scenarios are executing. Table 8-7 presents a summary of changes to the system properties in response to changes in the functional scenarios. Example properties that are added
to the e-exam system are presented in Figure 8-12. The properties “PO5” and “PO6” (shown in Figure 8-12.A) are added to the existing properties presented in Figure 8-5.A. These properties ensure that the “take training” feature is only available when it is selected. Figure 8-12.B shows two of the properties that need to hold while the new functional scenario “FS15” is executing. The property “PO1” specifies that the “suggest exams” interaction should precede the “select an exam” interaction when the user information is available, while the property “PO2” defines that a question must be viewed to a student only after it is provided (see Figure 8-12.B). The further details of the added system properties are described in Appendix “B”.

![Figure 8-12: Example properties that are added to the e-exam system](image)

![Table 8-7: Summary of the e-exam system’s properties](table)

<table>
<thead>
<tr>
<th>The Functional Scenarios</th>
<th>The System Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS1: Using the System by a Student</td>
<td>Local Properties: 7</td>
</tr>
<tr>
<td>FS7: Using the system by a Lecturer</td>
<td>Local Properties: 7</td>
</tr>
<tr>
<td>FS15: Take a Training</td>
<td>Local Properties: 2</td>
</tr>
<tr>
<td>FS16: Create a Student Account</td>
<td>Local Properties: 0</td>
</tr>
<tr>
<td>FS17: View an Exam’s Results</td>
<td>Local Properties: 0</td>
</tr>
<tr>
<td>FS18: Select and Exam</td>
<td>Local Properties: 4</td>
</tr>
<tr>
<td><strong>Total (After Changes): 74 Property</strong></td>
<td><strong>38</strong></td>
</tr>
</tbody>
</table>

8.2.4.2 Validating the Requirements’ Changes

To ensure the consistency of the adaptation scenarios and the validity of system variants after incorporating changes to the system scenarios, the e-exam system requirements are validated by the technique presented in Section 6.1.2. First, the number of adaptation conditions is identified.
The changed adaptation scenarios have 18 fragments. Four of these fragments have overlapping conditions as described in Appendix “B”, and then we have 14 conditions leading to 16384 ($2^{14}$) condition combinations.

Second, for each condition combination, an adaptation script is generated and checked (in the case that it is not already checked at the development time). The adaptation scenarios are designed properly, and then all scripts are consistent.

Third, using the consistent scripts, the system variants are generated (i.e. 16384 variants) and their properties are identified. Then, variants that are affected by the changed requirements are identified and transformed to Petri nets and their properties to CTL formulas to enable their validation using the Romeo tool (i.e. only the changed requirements are validated).

8.2.4.3 Updating the System Design Model

After validating the changes of the system scenarios (requirements), these changes need to be reflected to the system’s design model. To synthesize the evolved design model of the e-exam system (described below), we have used the technique introduced in Section 6.2.2.

*The System Functional Composite:* An updated model of the system’s functional composite is derived and shown in Figure 8-13. First, a context role that represents the student information and its provider are added. A contextual contract is also added between this role and the exams manager role (i.e. “CC2”), so that the exams manager can acquire the student information.

![Figure 8-13: The evolved functional composite of the e-exam system](image)

Second, a number of interactions are added to the contracts “FC1”, “FC4”, and “FC6”. These interactions are corresponding to functional messages that are added during the modification of
the functional scenarios. An example modified contract is “FC6” shown in Listing 8-7. In this contract, the interaction “suggest exams” (i.e. “i5”) is added.

**Listing 8-7:** Part of the modified functional contract “FC6”

<table>
<thead>
<tr>
<th>Functional Contract ID</th>
<th>FC6: Student_ExamsManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties:</td>
<td></td>
</tr>
<tr>
<td>Role A: Student;</td>
<td></td>
</tr>
<tr>
<td>Role B: ExamsManager;</td>
<td></td>
</tr>
<tr>
<td>Interaction Clauses:</td>
<td></td>
</tr>
<tr>
<td>i1: {StartExam(), AtoB, void};</td>
<td></td>
</tr>
<tr>
<td>i2: {EndExam(), AtoB, void};</td>
<td></td>
</tr>
<tr>
<td>i3: {ProvideQuestion (Question), BtoA};</td>
<td></td>
</tr>
<tr>
<td>i4: {ProvideAnswer (Answer), AtoB, void };</td>
<td></td>
</tr>
<tr>
<td>i5: {SuggestExams (ExamsList), BtoA};</td>
<td></td>
</tr>
<tr>
<td>Conversion Clauses (Temporal Constraints):</td>
<td></td>
</tr>
<tr>
<td>c1: {i4 response to i3}</td>
<td>c2: {i1 precedes i2 globally}</td>
</tr>
</tbody>
</table>

Third, existing behaviour processes are modified to reflect the changes in the scenarios. In addition, new behaviour processes are derived from the new functional scenarios and added to the functional composite. An example evolved process is presented in Listing 8-8 and visualized in Figure 8-14. This process is an updated version of the process shown in Figure 8-7, where new tasks, events, and constraints are added to reflect the changes in the scenario “FS1”.

**Figure 8-14:** The evolved behaviour process for a student’s interactions with the e-exam system
Listing 8-8: The description of the evolved process “P1: UsingSystemByStudent”

<table>
<thead>
<tr>
<th>Behaviour Process P1: UsingSystemByStudent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Event: LoginRequested;</td>
</tr>
<tr>
<td>End Event: Loggedout;</td>
</tr>
<tr>
<td>Tasks:</td>
</tr>
<tr>
<td>t1: {FR3.RequestLogin, Pre (LoginRequested), Post (RequestStudentInfo or NewUser)};</td>
</tr>
<tr>
<td>t2: {CreateAccount, Pre (NewUser), Post (RequestStudentInfo)};</td>
</tr>
<tr>
<td>t3: {StudentLogin, Pre (RequestStudentInfo), Post (LoginSuccess)};</td>
</tr>
<tr>
<td>t4: {TakeTraining, Pre (LoginSuccess), Post (TrainingTaken)};</td>
</tr>
<tr>
<td>t5: {TakeExam, Pre (LoginSuccess), Post (ExamTaken)};</td>
</tr>
<tr>
<td>t6: {ViewResult, Pre (LoginSuccess), Post (ResultViewed)};</td>
</tr>
<tr>
<td>t7: {FR3.Logout, Pre (ExamTaken or ResultViewed or TrainingTaken), Post (Loggedout)};</td>
</tr>
<tr>
<td>Temporal Constraints:</td>
</tr>
<tr>
<td>c1: {t3 precedes t4 globally}</td>
</tr>
<tr>
<td>c2: {t3 precedes t5 globally}</td>
</tr>
<tr>
<td>c2: {t3 precedes t6 globally}</td>
</tr>
</tbody>
</table>

The System Management Composite: The structure of the management composite for the e-exam system is still the same. But, there are a set of changes to its elements. First, new context attributes are introduced to the contract “CC1” to make the organizer role aware of the selection of the new system features. Second, a set of management actions are added to the management contract “MC1” to support the adaptation actions specified in the changed scenarios. Third, new rules that are corresponding to the new adaptation fragments are added to the organizer player. For example, two rules corresponding to the fragment “AD2” shown in Figure 8-11 are added to the organizer player as described in Listing 8-9.

Listing 8-9: Parts of the description for the changed organizer player of the e-exam system

<table>
<thead>
<tr>
<th>Adaptation Rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule EnableTakeTraining: {</td>
</tr>
<tr>
<td>When ValueChanges (TrainingSelected);</td>
</tr>
<tr>
<td>if TrainingSelected == True;</td>
</tr>
<tr>
<td>do AddBehaviourProcess (“P15”), AddTaskToProcess (“P1”, “TakeTraining”),</td>
</tr>
<tr>
<td>AddEventToProcess (“P1”, “TrainingTaken”);</td>
</tr>
<tr>
<td>Rule DisableTakeTraining: {</td>
</tr>
<tr>
<td>When ValueChanges (TrainingSelected);</td>
</tr>
<tr>
<td>if TrainingSelected == False;</td>
</tr>
<tr>
<td>do RemoveBehaviourProcess (“P15”), RemoveTaskFromProcess (“P1”, “TakeTraining”),</td>
</tr>
<tr>
<td>RemoveEventFromProcess (“P1”, “TrainingTaken”);</td>
</tr>
</tbody>
</table>

8.2.4.4 Realizing the Changes in the System’s Design Model

The changes to the design model of the e-exam system need to be realized either to new system instances or to a running system instance. First, in the case of taking the changes into account with new system instances, the design model is transformed to an executable ROAD model (as discussed in Section 5.4). Then, the ROAD model is used to create new instances of the e-exam system. Second, if the changes need to be realized to a running system, the technique introduced in Section 6.3 is used. The differences between the running system model and the new system’s
model are first computed. Then, adaptation actions that are corresponding to the differences are generated as an evolution script. Finally, these actions are applied to the running system. Part of the evolution script that represents the differences between the functional composites of the e-exam system shown in Figure 8-6 and Figure 8-13 is presented in Listing 8-10. This script can be applied to a running instance of the e-exam system to incorporate the unanticipated changes that are described above.

**Listing 8-10**: Part of the script for evolving the functional composite of the e-exam system

```java
//Adding new roles
organizer.addNewRole("CR2", "StudentInfo", "Role to represent student information");
...

//Changing the existing contracts
organizer.addNewTerm("FC6-I5", "SuggestExams", "void", "String#ExamsList", "BtoA", "FC6");
...

//Adding new facts
organizer.addNewFact("StudentInfo", "CompletedExams*StudentPerformance");
...

//Changing roles’ players binding
...

//Update defined processes
organizer.addProcess("BP15", "TakeTraining", "NotifyStudentInfo", "TrainingTaken");
organizer.addBehaviorUnit("BP15_BI1_NotifyStudentInfo");
organizer.addTaskToBehaviorUnit("BP15_BI1_NotifyStudentInfo", "FR4.NotifyStudentInfo_Task", "NotifyStudentInfo", "StudentInfoAvailable");
organizer.addBehaviorUnitToProcesDef("BP15", "BP15_BI1_NotifyStudentInfo");
...
```

8.3 Lessons Learnt from the Case Studies

In the previous sections, we have used our approach for developing and evolving two case study systems. In this section, we discuss the lessons learnt from doing these case studies. We classify the lessons into four groups based on the engineering tasks that need to be performed to develop and evolve such context-aware adaptive software systems.

**Requirements Specification**:

- *Specifying Complex Scenarios*: Using our approach, complex scenarios can be easily specified. To specify a complex scenario, a set of small scenarios are first specified. Then, the small scenarios are grouped to form the complex scenario by the sequence reference concept (e.g. the functional scenario “FS1” shown in Figure 8-9).

- *Incremental Requirements Specification*: In our approach, a system’s requirements are specified by defining a set of functional scenarios. Then, adaptation scenarios that define adaptations to the functional scenarios in response to context changes are
specified. Finally, a set of system properties that need to hold while the functional
scenarios are executing are specified (see Chapter 4). Thus, following our approach,
the system requirements are specified incrementally.

- **Specifying Context Information**: In our approach, the context information and its
  relationships with the system are represented explicitly. As such, the context and its
effect on the system can be clearly specified (captured).

- **Specifying System Properties**: A system’s properties can be easily specified in our
  approach, where they are represented in a form similar to the scenarios. In addition,
events of the properties are captured by messages specified in the system scenarios,
and then they can be easily understood by the stakeholders.

- **Specifying Local Properties**: A system’s local properties need to hold in specific
  context situations. Some of the local properties are related to the adaptation actions,
where a set of existence and absence properties are specified for elements that are
added or removed in response to context changes. The number of such properties is
large in systems that are highly dynamic. Therefore, specifying these properties is a
tedious task. As a future work, we will introduce a technique to enable the automatic
identification and specification of such properties using the adaptation scenarios.

- **Integrating Combined Fragments**: In some cases multiple combined fragments need
to be integrated. For example, a loop fragment can be part of an alternative fragment.
The alternative fragment may also be a part of another fragment. Thus, the graphical
representation of such integrated fragments becomes hard to manage. A solution to
that problem is to create small scenarios for such fragments and then integrate them
into the main scenario using the sequence reference concept.

**Requirements Validation**:

- **System Variants Generation**: Using our approach, a small set of scenarios specified
by the software engineer is able to cover a large number of system variants that suit
a large number of context situations. These variants are generated from the system
scenarios automatically (see Table 8-1).

- **Automatic Validation**: The automatic validation of the adaptation scripts (generated
from the adaptation scenarios) and the system variants (generated by applying the
scripts to the functional scenarios) helps to establish their consistency and validity.
**System Variants at Runtime:** In some cases, a large number of system variants are generated from the system scenarios. However, not all variants are used (required) by the users at runtime. Thus, validating the whole range of system variants is a time consuming task that may not be required. A solution to that problem is to allow the users’ involvement in the variants validation process, so that they can select variants that are only needed while the system in operation.

**System’s Design and Realization:**

- **Separation of Concerns:** Following our approach, a system is designed from three perspectives explicitly: functionality, context, and adaptive behaviour. In addition, elements of each perspective and their relationships with other system elements are explicitly specified. This separation of concerns enables the clear specification and manipulation of the system elements and their relationships.

- **Context Modelling:** The context model in our approach is simple, where the context information is captured as a set of entities that have a set of context attributes. Thus, a number of context aspects cannot be captured by the context model such as context uncertainty and dependency. In our approach, we assume that the context providers are responsible for capturing and managing such aspects. However, we will extend our approach to capture and manage these aspects in the future.

- **Design Synthesis:** Our approach enables the automatic synthesis of a system’s design model from its requirements (specified as a set of scenarios). This design model only contains two composites: functional and management. In a large scale system, these two composites become large. Therefore, the technique introduced for synthesizing a system’s design model needs to be improved, so that it can group similar scenarios and generate multiple composites corresponding to the groups. These composites are then used to form the overall system. Currently, the task of creating and grouping the multiple composites is performed manually by the software engineer.

- **Automatic Realization:** Following our approach, an executable ROAD model can be generated automatically from a system’s design model. Thus, the software engineer only works on models at a higher level of abstraction, while the lower level models are generated and deployed to the ROAD runtime environment automatically. When a ROAD model is deployed to the runtime environment, an instance of the system is created. This instance can be adapted in response to runtime context changes.
• Adaptation Rollback: In our approach, we support a system’s runtime adaptation in response to changes in the context information and the user needs by generating an adaptation script that is then applied to the system at runtime. But, the approach does not support the rollback of such adaptation when needed by the user. Therefore, we will extend our approach to support this feature in the future.

System’s Evolution:

• Evolution Patterns: To cope with unanticipated changes, we have used the change patterns described in Chapter 6. In evolving the travel guide and the e-exam systems, we found that all unanticipated changes are in the change patterns. Therefore, we can conclude that the change patterns cover the common unanticipated changes.

• Changes Validation and Realization: Following our approach, a small number of adaptation scripts and system variants are identified from the changed requirements which are then validated. As such, the validation process for the evolved model takes less time. In addition, changes to the system design are derived from its requirements and realized to the running system automatically.

• Design Evolution: In evolving a system’s design to cope with unanticipated changes, we found that the number of elements that form a composite’s structure increases in a slower rate in response to the increase in the number of scenarios. But, the number of behaviour processes has the same increase rate of the scenarios as shown in Table 8-1. Thus, we observe that the system’s structure is more stable than its behaviour as the structure does not change a lot in response to the unanticipated changes.

8.4 Summary

In this chapter, we have used our approach to develop and evolve two case study systems: the travel guide and the e-exam systems. The aim of performing these case studies is to demonstrate the approach’s applicability, and analyse its effectiveness in assisting the software engineer in performing the engineering tasks that are required for developing a software system that is able to cope with anticipated changes, and evolving the system while it is in operation in response to unanticipated changes.

The development and runtime evolution of the travel guide system are described throughout the thesis. Thus, we have summarized this case study by briefly describing the tasks that have been performed to specify and validate the system requirements, design the system based on its
requirements, realize the system by transforming the system’s design model to an executable ROAD model, and evolve the system at runtime to cope with unanticipated changes.

The travel guide software system is a small case study where it has a small number of system scenarios and variants. Therefore, we have performed another larger case study, i.e., the e-exam system. Similar to the travel guide system, we have used our approach to specify the functional and adaptation requirements of the e-exam system, validate the system requirements to ensure the adaptation requirements’ consistency and the conformance of the functional requirements to the system properties, synthesize the system’s design model from its scenarios and realize the system using this model, and evolve the system to cope with unanticipated changes.

Finally, we have discussed the lessons learnt from conducting the case studies. These lessons helped us to identify the strengths and limitations of using our approach to develop and evolve context-aware adaptive software systems.
Chapter 9: Evaluation

In this chapter, we evaluate our approach that assists the software engineer in the development and runtime evolution of context-aware adaptive systems. In order to evaluate the benefits and viability of the approach, we carry out four types of evaluations.

First, we perform a feature-based evaluation of the approach. In this evaluation, we list the features that an approach to developing and evolving context-aware adaptive software systems should support (as identified in Chapter 2), and for each feature we discuss how it is supported in our approach systematically. We also compare our approach with existing approaches relative to these features.

Second, to evaluate the effectiveness of the notations that have been introduced to enable a system’s requirements specification and its design, we use the physics of notations theory [203]. This theory has nine evidence-based principles to evaluate a modelling notation. We describe these principles and use them to evaluate the effectiveness of our notations.

Third, our approach aims to reduce the effort needed to develop and evolve context-aware adaptive systems by automating a set of engineering tasks (e.g. synthesis of a system’s design model from its requirements). To assess the reduction in a system’s development and evolution effort, we use COCOMO II [204] to estimate the efforts that are required to develop and evolve the software system when our approach is used and in the case that a traditional approach is used. Then, the difference between the estimated efforts is identified to quantify the reduction in the engineering effort that is gained by our approach. We also estimate the reduction in the time and cost that are needed for developing and evolving a context-aware adaptive system.

Fourth, performance evaluations for a software system developed in our approach and for the set of algorithms we have introduced in the approach are conducted. First, we assess the cost of adding the runtime adaptability feature to a software system (i.e. cost of changing the system while in operation). Second, we analyse theoretically the time complexity of the algorithms that are introduced for validating a software system’s requirements, synthesizing a system’s design model from its scenarios, updating a system’s design model to cope with unanticipated changes, and realizing the changes of a system’s design to a running instance of the system.

This chapter is organized as follows. In Section 9.1, we present a set of features that should be supported by our approach, and discuss how they are supported systematically. Evaluating the effectiveness of our graphical notations using the physics of notations theory is discussed in Section 9.2. In Section 9.3, we assess the reduction in the engineering effort that is gained when
our approach is used for developing and evolving a software system. Section 9.4 discusses the cost of adding the runtime adaptability feature to a software system, and the time complexity of the algorithms we have introduced in our approach.

9.1 Feature-based Evaluation of the Approach

To evaluate the appropriateness of our approach for developing and evolving context-aware adaptive systems, in this section, we perform a feature-based evaluation of the approach [205-206]. This feature analysis specifies to what degree our approach meets the requirements (or features) that need to be considered during the development and runtime evolution of context-aware adaptive software systems as described (identified) in Chapter 2.

9.1.1 The System Requirements Specification

To specify a context-aware adaptive system’s requirements, two types of requirements need to be captured: functional and adaptation. In addition, a set of qualities for the system’s context and functionality need to be specified.

(1) Specifying the Functional Requirements: The functional requirements of a context-aware adaptive system are a set of functions. These functions take the context information into account to give better suggestions to the end users. To specify these requirements, we have extended the UML sequence diagram as described in Section 4.1.2. Using the extended sequence diagram, the functional requirements are captured as a set of functional scenarios. In addition, we support the explicit representation of the context information in such scenarios using contextual lifelines (participants) and messages to specify the functional use of the context information.

(2) Specifying the Quality Requirements: To capture the quantifiable quality requirements [207-208], we have extended the “state invariant” concept of the UML sequence diagram (see Section 4.1.2). Using this extension, we can specify quantifiable qualities (e.g. response time as a functional quality, and freshness as a context quality) as a set of invariants (constraints) that need to hold during the system execution.

(3) Specifying the Adaptation Requirements: To capture the system adaptation in response to runtime context changes, a set of adaptation requirements need to be specified. In our approach, we have introduced the concept adaptation scenarios to capture such requirements. To specify these scenarios, we have extended the UML sequence diagram as discussed in Section 4.1.3. An adaptation scenario consists of a set of adaptation fragments that specify the system reaction(s) to context changes (notified by contextual messages).
(4) **Environment Uncertainty**: The system may be deployed into an environment that is not totally anticipated at the development time. This environment uncertainty affects the satisfaction of the system’s functional requirements [55]. To take the environment uncertainty into account, we enable runtime changes to the system requirements. Then, these changes are reflected to the system design and realization at runtime (see Chapter 6). Some of existing approaches represent the environment uncertainty during the development time by relaxing the system requirements (e.g. [19, 54]). As such, in the future, we will extend our approach to enable the relaxation of the functional scenarios to take into account the environment uncertainty at the development time.

Table 9-1 presents a comparison between our approach and the existing approaches that have been introduced to specify the requirements of context-aware adaptive systems.

**Table 9-1**: Comparing our approach with the approaches that support the requirements specification

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Requirements</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
</tr>
<tr>
<td>Quality Requirements</td>
<td>~</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F/C</td>
<td>(~)</td>
</tr>
<tr>
<td>Adaptation Requirements</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
</tr>
<tr>
<td>Environment Uncertainty</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>(~)</td>
</tr>
</tbody>
</table>

(−) Not Supported, (~) Partially Supported, (+) Fully Supported, (F) Functional Qualities, (C) Context Qualities.

9.1.2 Validating the System Requirements

In a large scale software system, the number of functional and adaptation scenarios is large. As such, the task of specifying these scenarios is tedious and error-prone. To ensure the validity of the system’s scenarios (requirements), two aspects need to be checked: the conformance of the functional scenarios to the expected system properties (which need to be first specified), and the consistency of the adaptation scenarios.

(1) **Specifying the System Properties**: While the system adapts from one variant to another at runtime, a set of system properties need to be preserved. These properties can be local (i.e. they are related to a specific system variant) or global (i.e. they need to hold in all system variants). To specify these properties, we support the graphical representation of the system properties in a form similar to the scenarios using the extension we have made to the UML sequence diagram as discussed in Section 4.1.4. In addition, we have extended the property specification patterns to specify context situations in which the local properties should hold.
(2) **Validating the Functional Requirements**: At runtime, the system has multiple variants and it switches between these variants in response to context changes. To ensure that the system properties still hold while the system adapts, we have introduced an algorithm to generate the system variants using the system’s adaptation scenarios, and to identify properties that need to be checked against them. The variants and the system properties are then transformed to formal models (i.e. Petri Nets and CTL formulas), so that the system properties can be checked against the variants formally using the Romeo tool as discussed in Section 4.2.2.

(3) **Ensuring Consistency of the Adaptation Requirements**: The adaptation requirements are defined as a set of scenarios to specify adaptation actions to be performed on the system to cope with context changes. Thus, the adaptation scenarios are consistent, if they generate a valid set of adaptation actions (as a script) in response to a context change. To ensure the consistency of the adaptation scenarios, we have introduced an algorithm (see Section 4.2.1) that parses these scenarios and generates possible adaptation scripts that can be fired at runtime. Then, each script is checked to ensure that it is free from a number of errors including adaptation actions’ conflict, redundancy, and incompleteness. Note that, a set of approaches have been introduced to ensure the context consistency (e.g. [25-26]). However, our approach does not support this feature, and then in the future we will integrate one of the existing approaches with our approach to ensure the context consistency.

Table 9-2 presents a comparison between our approach and the existing approaches that have been introduced to validating a system’s requirements.

**Table 9-2**: Comparison between our approach and approaches to validating the system requirements

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying the System Properties</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Validating the Functional Requirements</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Validating the Adaptation Requirements</td>
<td>Context Consistency</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Actions’ Validity</td>
<td>-</td>
<td>~</td>
<td>~</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

9.1.3 Designing and Realizing the System

To design a context-aware adaptive software system, the system’s requirements need to be mapped to a system’s design model (i.e. moving from the system’s problem space to its solution space). This design model specifies elements that represent the system’s functionality, context
model, and adaptive behaviour. The design model also needs to have a set of constraints that are corresponding to system properties that need to hold at runtime. In addition, a causal connection between the system requirements and its design model needs to be maintained. Furthermore, the representation of the system’s executable elements needs to be flexible to enable the runtime system adaptation.

(1) **Designing the System Functionality:** To specify the system functionality, we have introduced the system’s functional composite (see Section 5.2.1). This composite contains a set of functional roles (an abstract representation of the system’s functions) that interact with each other through functional contracts. In addition, the composite has functional players that play these roles to provide the actual functionality at runtime. Furthermore, to specify the functional behaviour, the composite has a set of event-based processes (i.e. a process is defined as a set of loosely coupled tasks that are related to each other by events). Each behaviour process defines in what order a set of simple tasks are executed to provide a composite (complex) task.

(2) **Capturing the Context Model:** The context information is used by the system to continue its operations or can trigger the system’s runtime adaptation. Thus, in our approach the system’s functional and management composites have context models to define the context information that is needed by the system functionality and management. To specify the context model, we have introduced three main concepts: context role, context provider, and contextual contract as described in Section 5.2.1. The context roles specify context entities that the system is interested in, while the context providers provide the context attributes of these entities. The contextual contracts are used for defining the context attributes that are required by the system’s functional and management roles.

(3) **Capturing the System’s Adaptive Behaviour:** To capture the system’s adaptive behaviour, we have introduced the concept of management composite (see Section 5.2.2). This composite has an organizer role that is bound with a player that uses a set of event-condition-action (ECA) rules to decide the system adaptation in response to runtime context changes. In addition, it has a management contract that specifies the adaptation actions that can be applied to the functional composite at runtime. Furthermore, the system’s functional composite is represented inside the management composite by the functional system role.

(4) **Specifying the System’s Temporal Constraints:** While the system in operation a number of temporal constraints need to be preserved to ensure that the system works properly. To define such constraints, we have used the IRS language [188]. The constraints can be related to binary interactions between two system roles, and then they are specified into the functional contracts.
If the constraints are related to interactions that are specified between more than two roles, the constraints are specified within the behaviour processes (see Section 5.2.1).

(5) **Design Validation:** To validate a system’s design, we have used the technique introduced by the ROAD framework [140]. To use this technique, we transform a system model designed in our notation to a ROAD model (see Section 5.4). The ROAD model is then validated by the ROAD framework. The validation module of the framework only checks the system behaviour against the temporal constraints. Thus, as a future work, we will introduce a technique to enable the validation of the system structure to ensure that the structure is well formed.

(6) **Managing the System’s Complexity:** To manage the system complexity, we have used the “separation of concerns” and “levels of abstraction” concepts. First, a context-aware adaptive system contains multiple aspects including functionality, context, and adaptive behaviour. Thus, these aspects need to be explicitly represented to ease the task of designing the system. In our approach, we achieve the “separation of concerns” by modelling the system as two composites that capture the system functionality and management separately. In addition, in the functional composite, the functional behaviour is modelled separately from its structure. Furthermore, the system’s context models which are specified inside the two composites are kept as separate as possible through the context roles and the contextual contracts (see Sections 5.2.1 and 5.2.2). Second, the system can be modelled at different levels of details. The system is represented as two main composites at the higher level, and then the functional players and context providers at this level are modelled as sub-systems at the lower level. In addition, the lower level can be a higher level for another lower level (i.e. a hierarchical composition of the system). Furthermore, in the behaviour model, a composite task can be defined in a process to hide the details of a set of related tasks that are designed into another separate process (i.e. a process can be viewed as a collection of sub-processes).

(7) **Maintaining a Causal Connection between the System Requirements and Design:** The system design needs to be kept synchronized with the system requirements to ensure that all requirements have been considered during the system design. To maintain a causal connection between the system requirements and its design model, we have introduced a set of algorithms in Section 5.3. These algorithms automatically synthesize the system’s design model from its requirements (specified as two sets of scenarios). The algorithms reduce the effort required by the software engineer to design the system, where the system’s initial design is generated. Then, the synthesized model can be further modified by the engineer to add elements that are related to the system’s solution space but cannot be synthesized from the scenarios (e.g. specifying a set of players for a functional role).
(8) **System Realization**: To assist the developers of a context-aware adaptive system, part of the system implementation can be generated from its design model. In our approach, we support this feature by transforming our design model to an executable ROAD model as described in Section 5.4. When this model is deployed to the ROAD runtime environment, an instance of the system is created. In addition, the different aspects (including functional structure, functional behaviour, and context model) of this instance can be adapted at runtime in response to context changes. To have a fully functioning system, the developers only need to develop (or find) a set of players and context providers and bound them with the instantiated system’s functional and context roles respectively.

(9) **System Flexibility**: To enable the system’s runtime adaptation, the system elements need to be easily manipulated (adapted). In designing a context-aware adaptive software system, we follow an organizational approach (see Section 5.1). In such an approach, the relationships (i.e. different type of contracts) between the system elements are represented explicitly. Therefore, the system elements and their relationships can be easily manipulated. In addition, the system’s functional players and context providers are separated from its roles, and then they can be bound or unbound from the roles at runtime. Furthermore, the system behaviour is designed as event-based processes where each process is modelled as a set of loosely coupled tasks that are related to each other by events that specify the pre and post conditions of the tasks. Thus, a process can be easily changed by modifying the pre and post conditions of its tasks.

**Table 9-3**: Comparing our approach with approaches that support the system design and realization

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Design</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Functionality</td>
<td></td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S/B</td>
</tr>
<tr>
<td>Context Model</td>
<td></td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E/B</td>
</tr>
<tr>
<td>Adaptive Behaviour</td>
<td></td>
<td>A</td>
<td>A</td>
<td>T</td>
<td>U</td>
<td>C</td>
<td>M</td>
<td>A</td>
<td>M</td>
<td>U</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>System Qualities</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F/C</td>
</tr>
<tr>
<td>Model Constraints</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Design Validation</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Managing the Complexity</td>
<td></td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Design Synthesis</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>System Realization</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Realization</td>
<td></td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>Change Time</td>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

Table 9-3 presents a comparison between our approach and the existing approaches that have been introduced to designing and realizing context-aware adaptive software systems.

9.1.4 The System Evolution

During the lifetime, a context-aware adaptive system needs to evolve to consider unanticipated changes in its environment and functionality. To cope with these changes, a set of tasks need to be performed. First, changes to the functional and adaptation requirements need to be specified. Second, to ensure that the adaptation requirements are still consistent and the functional requirements are still valid when the requirements are changed, the requirements specification needs to be re-validated. Third, the changed requirements need to be reflected to the system’s design by adding and changing a set of elements to take the requirements’ changes into account. Fourth, a system instance may be in operation and the provider needs to reduce the downtime of this instance to increase the customers’ satisfaction. As such, the changes to the design model of the system need to be realized to the running system instance.

(1) Specifying the System Requirements’ Changes: The changes to the system requirements can be to its functional and/or adaptation requirements. To specify the functional requirements’ changes, we support the engineer in modifying the system scenarios to cope with such changes by introducing new functional scenarios or by modifying existing ones as presented in Section 6.1.1. The changes to the functional scenarios are triggered by the need to include new context information or functionality, or to modify the system functionality and its context model. In the same manner, to cope with changes in the adaptation requirements, exiting adaptation scenarios are modified or new scenarios are introduced (see Section 6.1.1). The main reason to change such type of scenarios is to consider new adaptive behaviour or to change the system’s reactions to anticipated context changes.

(2) Re-validating the System’s Requirements: After a set of changes have been introduced to the system’s scenarios (by adding new scenarios or modifying existing ones), the scenarios need to be re-validated. To reduce the time needed for validating the changed requirements, we have introduced a technique that takes into account the results of the requirements validation prior to the change. In this technique, we only validate new and changed adaptation scripts to ensure the consistency of the adaptation requirements. In addition, we only check new and changed system variants against the relevant system properties as discussed in Section 6.1.2.

(3) Changing the System Design to Cope with Unanticipated Changes: The system’s design model needs to evolve to take into account unanticipated changes in the system requirements.
To assist the software engineer in evolving the system’s design model (in the case that he wants to change the system’s design model directly), we have introduced a set of evolution (change) patterns. These patterns specify the required changes to the system’s design model to consider the changes in the system’s context (Patterns 1 to 4 in Section 6.2.1.2), functionality (Patterns 5 to 7 in Section 6.2.1.3), and adaptive behaviour (Patterns 8 and 9 in Section 6.2.1.4).

(4) Reflecting the Requirements Changes to the System’s Design Automatically: The changes to the system’s scenarios (requirements) may be small or they are performed several times. In such cases, synthesizing the whole design model of the system from its scenarios is inefficient. To tackle this problem, we have introduced a technique in Section 6.2.2. This technique detects the type of requirements’ change. Then, it selects a proper change pattern to be applied into the system’s design model to consider such change. Following this technique, the system’s design and its scenarios evolve hand in hand.

(5) Realizing the System’s Design Changes to a Running System: While an instance of the system is running, the provider may want to evolve that instance without stopping it (to increase the customers’ satisfaction, for example). To enable the system’s runtime evolution, we have introduced a technique in Section 6.3 that computes the differences between a running system’s model and its evolved model. Then, these differences are used to generate a script that contains a set of adaptation actions to realize the changes of the design model. Finally, the generated script is executed to evolve the running system. A set of players and context providers may also be developed (if needed) and bound to the system’s roles, so that the system can deal with the changes that were not anticipated at the development time.

Table 9-4 presents a comparison between our approach and the existing approaches that have been introduced to evolve context-aware adaptive systems to cope with unanticipated changes while they are in operation.

**Table 9-4:** Comparison between our approach and the approaches that enable the system evolution

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Specifying Changes</th>
<th>Design Model</th>
<th>Synthesis of the Design Changes</th>
<th>Validating the Changes</th>
<th>Realizing the Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Specification</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Design Model</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Synthesis of the Design Changes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Validating the Changes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Realizing the Changes</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Cote et al. [41]</th>
<th>Gomaa et al. [79]</th>
<th>Gozzi et al. [155]</th>
<th>Ahmad et al. [157]</th>
<th>Morin et al. [28]</th>
<th>Traditional Validation (e.g. [67])</th>
<th>Incremental Validation (e.g. [67])</th>
<th>Oreizy et al. [33]</th>
<th>Andrade et al. [77]</th>
<th>MUSIC [34]</th>
<th>ROAD [62]</th>
<th>Dynamic Update (e.g. [160-161])</th>
<th>The Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.2 Effectiveness of the Approach Notations

In the thesis, we have introduced a set of visual notations to enable the system’s requirements specification and design. In this section, we evaluate the effectiveness of these notations. We first present the physics of notations theory [203, 209] that has been introduced to evaluate a modelling notation. Then, we use the theory to evaluate our notations.

9.2.1 The Physics of Notations Theory

The physics of notations theory provides a framework to evaluate visual notations of modelling languages using nine evidence-based principles [203, 209]. These principles are derived from various scientific disciplines such as perceptual psychology, graphic design, human computer interface, linguistics, and communication. The theory principles have been used to evaluate the effectiveness of the UML [210], i* [211], and BPMN [212]. In the following, we describe these principles.

Semiotic Clarity: The “semiotic clarity” principle means that there is a one-to-one correspondence between a language’s constructs and its graphical symbols [213]. To ensure the semiotic clarity of a modelling language, it should be free from four types of errors: symbol deficit, symbol overload, symbol redundancy, and symbol excess. The symbol deficit occurs when a language construct do not have a corresponding graphical symbol. When two language constructs are represented by the same graphical symbol, the symbol overload problem occurs. The symbol redundancy occurs when two graphical symbols are used for a language construct. When a graphical symbol do not have a corresponding language construct, the symbol excess problem occurs.

Perceptual Discriminability: A language’s symbols need to be easily distinguishable from each other [214]. Discriminability of the language symbols is measured by the visual distance between them. The visual distance is determined by the number of visual variables in which the symbols differ and the size of these differences. The visual variables include shape type, symbol size, symbol colour, and shape style (texture), etc.

Semantic Transparency: Semantic transparency is defined as to what extent the symbols meaning can be inferred from their graphical representations [215]. A symbol is a semantically immediate if a novice reader can understand the symbol meaning from its appearance alone. When a symbol has an arbitrary relationship with its meaning, it is a semantically conventional. A symbol is a semantically preserve if its appearance makes the reader infers a different (or an opposite) meaning.
Complexity Management: The “complexity management” principle ensures that a modelling language is able to deal with the diagrammatic complexity. The complexity is measured by the number of elements that can appear in one diagram [216-217]. To manage the complexity of a diagram, modularity and hierarchical structuring concepts can be used.

Cognitive Integration: In the case of a large scale system that needs to be modelled using a number of diagrams. The modelling language should support the integration of such diagrams, so that a reader can have a general understanding (view) of the system model (i.e. cognitive integration)[218-219].

Visual Expressiveness: The “visual expressiveness” principle measures to what extent the graphic design space has been used to specify a modelling notation (i.e. the number of visual variables used in constructing a notation’s graphical symbols and the values of these variables). The “perceptual discriminability” principle focuses on pair-wise variations between graphical symbols, while the “visual expressiveness” principle measures the variations between the entire notation symbols.

Dual Coding: According to the “perceptual discriminability” and “visual expressiveness” principles, text is not useful to encode information in a visual notation. However, it can be used as a supplement to graphical symbols where the use of text and graphic (i.e. dual coding [220]) is more effective in conveying information than using one of them only.

Graphic Economy: Graphic complexity is measured by the number of graphical symbols in a modelling notation [221]. The notation designer can have unlimited number of graphical symbols by combining visual variables in different ways. But, there is a limit on the number of elements that can be easily recognized by human and beyond this each new symbol reduces the notation effectiveness. The human ability to differentiate between alternatives is a round six categories [222]. Thus, the number of symbols in a notation needs to be around six symbols.

Cognitive Fit: The cognitive fit principle states that there should be a three-way fit between audience (i.e. sender and receiver), task characteristics (i.e. for what purpose a notation is used), and a medium on which the notation is represented [223]. In practice, this principle is analysed using expert-novice differences and differences in representational medium. First, with respect to expert-novice differences, a notation should consider its audience where a notation with few symbols is used for novice readers and a notation with clear and comprehensive symbols is used for experts to capture all details. Second, a notation’s graphical symbols need to suit different representational mediums, where symbols that drawn by computer-based tools should be also drawn on a white-board easily.
9.2.2 Evaluating the Notation for Specifying the System Requirements

To specify the requirements of a context-aware adaptive system, two types of requirements need to be captured: functional and adaptation. In addition, a set of properties that should hold while the system in operation need to be specified. To specify the system requirements and properties, we have introduced a graphical notation that extends the UML sequence diagram (see Chapter 4). In the following, we evaluate this notation.

**Semiotic Clarity:** In Chapter 4, we have presented a set of meta-models as extensions to the UML sequence diagram to capture a system’s requirements and its properties. The total number of concepts (i.e. the language constructs) in the meta-models are 22. These concepts have their corresponding graphical symbols, where for each meta-model concept a graphical symbol is introduced (i.e. a one-to-one correspondence between the concepts and the graphical symbols). Therefore, our notation is free from symbol’s deficit, overload, redundancy, and excess.

**Perceptual Discriminability:** A system’s requirements and properties are captured as a set of scenarios. The scenarios are specified by five concepts: “lifeline”, “message”, “interaction use”, “state invariant”, and “combined fragment”. The lifeline is represented as a rectangle with a line descending from it, while the message is captured as an arrow line. A rectangle with the word reference on its top left and a sequence name represent the interaction use. The state invariant is captured as a round rectangle. The combined fragment is represented as a rectangle with the combined fragment type on its top left and a set of messages grouped by the fragment. The five concepts are also specialized for certain aspects. For example, there are three types of lifelines that are represented as variations to the basic form of a lifeline by having different border styles. The five concepts and their specializations are distinguishable from each other as either they have different shapes or they use a shape with different border styles. But, the visual distance between these concepts is small, in particular in the case of concept specialization. In designing our notation, we were restricted by the notation of the UML sequence diagram that is widely accepted in practice as we have extended this notation to capture our concepts. Thus, we inherit this drawback from the UML sequence diagram.

**Semantic Transparency:** The graphical symbols of our notation to specifying a system’s requirements and its properties are represented as a set of boxes and lines. Thus, such symbols do not convey any information about their meaning (i.e. their meaning is purely conventional and needs to be learnt). However, in general, the concept of scenario can be easily inferred from a diagram, where there are a set of participants that interact with each other through messages, and the order of the messages is from top to bottom.
**Complexity Management:** To specify the requirements for a large scale software system, the “interaction use” (i.e. the sequence reference) concept is used. Using this concept, the system is specified as a set of scenarios that are smaller in size. Then, these small scenarios are grouped using the interaction use to form larger scenarios (i.e. specifying the system with different levels of abstraction details). In addition, in our approach, the system requirements and properties are captured in three different groups of scenarios, so that they can be easily managed.

**Cognitive Integration:** Following our approach, the system requirements are specified as functional and adaptation scenarios. The adaptation scenarios are about changing the functional scenarios. In addition, the system properties are specified on the functional scenarios. To enable the integration of these three views, we have specified identifiers for the functional scenarios’ elements. These identifiers are then used by the adaptation scenarios and the system properties, so that relationships between the functional and adaptation scenarios and between the functional scenarios and the system properties are explicitly represented.

**Visual Expressiveness:** To represent the meta-models’ concepts, we have introduced a set of graphical symbols that use a number of visual variables. The visual variables include horizontal and vertical positions, shape, texture, and colour (as discussed in Chapter 4). However, we did not use visual variables such as brightness, orientation, and size which limit the graphic space of our notation design.

**Dual Coding:** To increase the effectiveness of conveying information about our notation, we supplemented the notation with text. This text describes elements’ types, where we specify an identifier for each element type (e.g. the identifier “FM” means a functional message). As such, the types of elements can be easily inferred from their identifiers.

**Graphic Economy:** The notation to specifying a system’s requirements includes 22 graphical symbols which are over the human ability to differentiate between symbols as the normal number of symbols is around six. However, we have five basic concepts and the other concepts are extensions (specialization) to them. Therefore, the reader needs to only know the type of an element relative to the five (general) concepts. Then, he can identify the element’s special type. For example, the reader can specify an element type as a lifeline, and then he can identify the lifeline type (e.g. functional). In this way, the reader needs to only distinguish between less than six types of elements at a time.

**Cognitive Fit:** The cognitive fit is analysed using expert-novice differences and differences in the representational medium. First, the intended audience of our notation is the stakeholders. Therefore, we support the specification of the requirements as a set of scenarios which can be
conceived easily by the stakeholders. In addition, the diagrams produced by the notation are kept simple by the “complexity management” and “cognitive integration” principles. Second, our graphical symbols are simple geometrical shapes. As such, they can be represented using a computer-based tool (as described in Chapter 7). In addition, they can be easily drawn by hand which is a convenient way for making diagrams using a white-board and paper sketches.

9.2.3 Evaluating the Notation for Designing the System

To design context-aware adaptive software systems, we have introduced an organization-based meta-model. We have also introduced a graphical notation to be used by the software engineer in designing a context-aware adaptive system following this meta-model (see Chapter 5). In the following, we evaluate this design notation.

Semiotic Clarity: The organization-based meta-model to enable a system’s design is divided into two main composites: functional and management. The functional composite is divided into functional structure and functional behaviour. The functional structure is constructed using 7 concepts, while the functional behaviour is formed using 8 concepts as presented in Chapter 5. The management composite is also constructed using another 4 concepts (see Chapter 5). These 19 concepts have their corresponding graphical symbols, and there is a one-to-one mapping between the concepts and the graphical symbols. Thus, the symbols of our notation to design a context-aware adaptive software system are free from symbol deficit, symbol overload, symbol redundancy, and symbol excess.

Perceptual Discriminability: Following our approach, the system is designed using seven concepts that have their corresponding graphical symbols: “composite” represented as an oval, “role” captured as a round rectangle, “player” specified as a parallelogram, “contract” captured as a diamond, “event” represented as a hexagon, “task” captured as a rectangle, and “process” as a set of tasks and events that are linked by lines. In addition, some of the concepts are extended to capture special type of elements. For example, there are three contract types that are captured by diamonds with different border styles. The graphical symbols that have been introduced are from the same family of shapes except the composite symbol. In addition, variations between some symbols are small (e.g. border style). Therefore, the visual distance between the notation symbols is small, and then the graphical symbols need to be improved by having symbols from different families of shapes with greater degree of variations.

Semantic Transparency: The graphical symbols to design a context-aware adaptive system are simple shapes that are connected by a set of lines. Thus, a reader cannot infer any meaning
from these symbols which means the symbols are conventional and need to be learnt. However, the reader can infer the basic elements of the system and how they are connected to each other generally. He also may infer the way in which the system executes to achieve complex tasks by a set of behaviour processes.

**Complexity Management:** To design a large scale software system, our approach enables the system modelling at different levels of abstraction. First, from the system’s structure point of view, a functional player or a context provider can be designed as another composite (i.e. a role player/provider is a subsystem). Thus, at a lower level, the system is designed as a set of small composites (subsystems), and these composites are grouped at a higher level to form the overall system. Second, a behaviour process can be complex. To design such complex process, a set of small processes can be designed. Then, these processes are represented as composite tasks in the complex process. Thus, the complex process is represented using a small number of composite tasks that are detailed in other behaviour processes.

**Cognitive Integration:** Following our approach, the system is designed as two composites: functional and management. In addition, the functional composite has the system’s functional structure and behaviour. Therefore, the system is designed using three perspectives: functional structure, functional behaviour, and management. To keep these views integrated in the reader mind, we first make the functional composite as player in the management composite where the organizer role can request a set of adaptation actions to be performed on the system’s functional composite (i.e. the management composite is an integrated view of the system). Second, in designing the functional composite, the composite elements and the interactions between them are specified. Then, a set of behaviour processes are defined based on the interactions between the functional elements (i.e. the functional composite is an integrated view of the functional structure and behaviour of the system).

**Visual Expressiveness:** The graphical symbols of the system’s design notation have used a number of visual variables such as vertical position (in designing a behaviour process), shapes, textures, and colours. But, we did not consider other visual variables such as horizontal position, size, brightness, and orientation in constructing the notation symbols. As such, the design space for our notation is limited.

**Dual Coding:** Using text we have specified the details for a number of graphical symbols in the system’s design model. For example, a set of interactions between the system’s functional roles are specified textually in functional contracts. In the same manner, a set of adaptation rules
are specified in the system’s organizer player (i.e. the text has been used to convey information that cannot be conveyed using the graphical symbol or the textual elements alone).

*Graphic Economy:* We have 19 graphical symbols to enable the design of a context-aware adaptive system. This number of symbols is over the human ability to easily identify graphical symbols. However, similar to the notation for specifying the system requirements, the graphical symbols to design a system have seven basic concepts while the others are extensions to them. Therefore, the reader can easily identify an element in a two-step process. In the first step, the element is in one of the basic concepts, while the second step is used for identifying the special type of the element based on its variation from the basic concept.

*Cognitive Fit:* To analyse the cognitive fit of the notation, we use expert-novice differences and differences in the representational medium. First, the intended audience of the notation is the software engineers (i.e. expert users). Therefore, we have provided a comprehensive set of symbols with clear semantics. In addition, diagrams that can be produced by these symbols are kept simple by following the “complexity management” and “cognitive integration” principles (see above). Furthermore, we have used text to specify the required details of the design model. Second, a set of simple geometrical shapes are used in designing the notation. Thus, they can be easily represented by a computer-based tool (as shown in Chapter 7), or drawn by hand to make diagrams using a white-board and paper sketches.

### 9.2.4 Summary

Table 9-5 summarises the evaluation of our notations for specifying the system requirements, and designing the system using the physics of notations theory.

<table>
<thead>
<tr>
<th>Theory Principles</th>
<th>Requirements Specification</th>
<th>System Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semiotic Clarity</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Perceptual Discriminability</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Semantic Transparency</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Complexity Management</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cognitive Integration</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Visual Expressiveness</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>Dual Coding</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Graphic Economy</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cognitive Fit</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

(~~) Partially Considered (+++) Fully Considered
9.3 Reduction in Engineering Effort

To develop and evolve a context-aware adaptive software system by our approach, the software engineer first specifies two sets of scenarios that capture the system’s functional and adaptation requirements and a set of properties that need to hold while the system in operation. Second, the requirements specification is validated to ensure the consistency of the adaptation requirements, and the validity of the functional requirements relative to the expected system properties. Third, the system’s design model is synthesized from its requirements (scenarios). The design model is then modified by the engineer to add elements that are related to the system’s solution space but cannot be synthesized from the scenarios (e.g. adding functional players). Fourth, to realize the system, the system’s design model is transformed to an executable ROAD model which is then deployed to the ROAD runtime environment to create a system instance. To make the system fully functioning, the system developers need to develop (or find) functional players and context providers and bind them with the running system’s roles. Finally, in response to unanticipated changes, the software engineer modifies the system’s scenarios. Then, changes to the scenarios are validated, and reflected to the system design and the running system automatically.

Using our approach, a number of engineering tasks are automated to reduce the engineering effort that is required for developing and evolving context-aware adaptive software systems. In the following, we analyse sources of effort reduction that can be obtained by performing these tasks in our approach. Then, the constructive cost model (COCOMO)II [204] is used to estimate the reduction in the engineering effort.

9.3.1 Sources of Effort Reduction

In this section, we analyse how our approach reduces the engineering effort that is required for specifying and validating a software system’s requirements, designing and realizing the system, and evolving the system at runtime in response to unanticipated changes.

9.3.1.1 Specifying the System Requirements and Properties

To specify a context-aware adaptive software system’s requirements, we have introduced a technique in Chapter 4. It captures the system requirements as two sets of scenarios: functional and adaptation. The functional scenarios capture the system functionality, while the adaptation scenarios specify runtime changes to the system functionality in response to context changes. In addition, the system properties that need to hold at runtime are specified in a form similar to the scenarios (see Chapter 4).
The task of specifying the system requirements and properties needs to be performed by the software engineer manually. Thus, our approach does not reduce the engineering effort required for doing that task. However, using the developed tool, the software engineer first specifies the functional scenarios. Then, during the adaptation scenarios specification, the tool suggests to the engineer the adaptation actions that can be performed on the functional scenarios. In the same manner, the tool suggests to the engineer the events that can be used in specifying the system properties (using the messages specified in the functional scenarios). Thus, the effort required to specify the adaptation scenarios and the system properties is reduced.

9.3.1.2 Automatic Validation of the System Requirements

In Chapter 4, we have introduced a technique to validate a system’s functional and adaptation requirements. First, to check the consistency of the adaptation requirements, a set of adaptation scripts are generated from the adaptation scenarios. Such scripts specify the different adaptation actions that can be applied to the system in response to context changes. These scripts are then checked for errors such as redundancy, incompleteness, and conflict of their adaptation actions. Second, to validate the functional requirements, a set of system variants are generated from the functional scenarios and the properties to be checked against these variants are identified. The variants and the properties are then translated to formal models to enable their formal validation using the Romeo tool.

Following our approach, the system requirements are validated automatically with limited involvement from the software engineer, where he only needs to analyse the validation results and modify the system scenarios when needed based on these results. Therefore, a little effort is required from the engineer to validate the system requirements. In addition, the engineer does not need to have a profound knowledge of formal methods as he works on the system scenarios and the formal models are automatically generated and validated using our tool with limited involvement from him.

9.3.1.3 Synthesis of the System’s Desing Model from its Requirements

A system’s design can be automatically synthesized from the system requirements (specified as scenarios) using the set of algorithms presented in Chapter 5. In general, the system’s functional scenarios are transformed to the system’s functional composite, and the management composite is derived from the adaptation scenarios. For each composite, a set of roles are derived from the scenarios’ participants and the connections between the roles (as contracts) are synthesized from the messages that are exchanged between the scenarios’ participants. In addition, skeletons of players and context providers are generated and bound to the synthesized roles.
Using the technique introduced in Chapter 5, the system’s design model is generated from its requirements (specified as scenarios) automatically. Then, a small number of modifications are performed on the design model to added elements that are related to the system’s solution space and cannot be derived from the scenarios. These modifications include specifying players and context providers, types of interactions’ parameters, and constraints on the management contract between the organizer role and the functional system role. As such, the effort required from the software engineer to design a context-aware adaptive system is small, where a small number of modifications to the design model (synthesized from the system scenarios) are only required.

9.3.1.4 Generating the System Implementation from the Design Model

To realize a software system designed in our approach, we transform the system’s design model to an executable ROAD model (discussed in Chapter 5). The executable model is then deployed to the ROAD runtime environment to create an instance of the system. At runtime, in response to context changes, a set of adaptation actions are applied to this system instance. These actions are decided by the organizer player that is generated from the system’s design model.

To have a fully functioning system, skeletons for functional players and context providers that are generated in the design model (as described in Chapter 5) need to be completed by the system developers. As such, the effort required to realize a software system using our approach is limited to the amount of code that needs to be written to complete the functional players and the context providers, while the code required for enabling the interactions between the system roles, executing the behaviour processes, and adapting the system is automatically generated and deployed to the ROAD runtime environment. In addition, if a functional player or a context provider is already exists, it can be easily used in our approach. To do so, the developer needs to write a small amount of code that calls the methods (operations) of an existing functional player (or a context provider) in the empty bodies of methods generated by our approach. This further reduces the effort needed to realize context-aware adaptive software systems.

9.3.1.5 Automatic Realization of the System’s Runtime Changes

To cope with unanticipated changes, we have introduced a set of techniques in Chapter 6. First, the unanticipated changes are incorporated to the system scenarios by introducing new scenarios or modifying existing ones. Second, the requirements’ changes are validated to ensure that the adaptation requirements are still consistent, and the changed system variants are still valid with respect to the system properties. Third, the requirements’ changes are automatically reflected to the system’s design model by the change patterns described in Chapter 6. Finally, the changes to the system’s design are realized to the running system by computing the differences between the
running system model and its evolved model. Then, adaptation actions corresponding to these differences are generated and executed to evolve the running system.

Following these techniques, the software engineer only performs the changes to the system scenarios using the change patterns. Then, the scenarios changes are automatically validated and reflected to the system’s design model. He may also perform some modifications to the updated design model to add elements that cannot be derived from the scenarios. Thus, the effort needed from the software engineer is small, where only a few changes to the system scenarios and the design model are required. To realize the changes of the design model, the changes are reflected to the running system automatically. Thus, the system developers need to only write the amount of code for completing the generated players and context providers (if needed) without worrying about how the changes to the system structure and its behaviour are realized.

9.3.1.6 Summary

To develop and evolve a context-aware adaptive software system in our approach, a number of engineering tasks need to be performed. These tasks are performed either manually with support from the developed tool (i.e. requirements specification), or automatically with limited human involvement to perform certain aspects that cannot be performed in an automated manner (e.g. designing the system).

As discussed above and summarised in Table 9-6, all engineering tasks are automated except the requirements specification and its modification in response to unanticipated changes. Thus, our approach reduces the effort required for the development and runtime evolution of context-aware adaptive systems, where most of the tasks are performed in an automated manner with limited involvement from the software engineer (see Table 9-6).

Table 9-6: Summary of the engineering tasks and how they are performed in our approach

| Engineering Task                        | How it is Performed?
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements Specification</td>
<td>Manually</td>
</tr>
<tr>
<td>2. Requirements Validation</td>
<td>Automated</td>
</tr>
<tr>
<td>3. Designing the System</td>
<td>Automated</td>
</tr>
<tr>
<td>4. Realizing the System</td>
<td>Automated</td>
</tr>
<tr>
<td>5.1 Specifying Requirements Changes</td>
<td>Manually</td>
</tr>
<tr>
<td>5.2 Validating the Changes</td>
<td>Automated</td>
</tr>
<tr>
<td>5.3 Updating the Design Model</td>
<td>Automated</td>
</tr>
<tr>
<td>5.4 Realizing the Changes</td>
<td>Automated</td>
</tr>
</tbody>
</table>
9.3.2 Estimating the Effort Reduction

To estimate the effort reduction that is obtained by our approach in developing and evolving a context-aware adaptive system, we use the constructive cost model (COCOMO) II [204] which is a widely accepted and a well documented software cost model [224]. We first use COCOMO II to estimate the required efforts for developing and evolving a software system (e.g. the travel guide software system) with and without our approach. Then, the effort reduction is computed as the difference between the estimated efforts.

9.3.2.1 Background: COCOMO II

The COCOMO II model allows the estimation of the development and maintenance effort of a software system. It includes three sub-models: application composition, early design, and post-architecture [204]. These models are used based on the amount of information that is available about the development lifecycle (to support the effort estimation). The post-architecture model is the most mature and detailed model among the three sub-models, and is used for estimating the effort of a software system’s development and maintenance when the software lifecycle is established. It also has been further calibrated to provide accurate effort estimation [225]. Thus, we use the post-architecture model to estimate the effort for the development and maintenance (i.e. evolution in this thesis context as discussed below) of a context-aware adaptive system.

The post-architecture model is used for estimating the effort in person months (PM) and the time (TDEV) for developing a software system. First, to estimate the effort (PM) required for developing a software system in person months, Equation 1 is used (see below). This equation specifies the effort needed to develop the system that have a size measured in thousand lines of code (KLOC) with respect to seventeen effort multipliers (“EM”), and five scale factors (“SF”) which are used to calculate the exponent (“E”). The effort multipliers and the scale factors are calibrated from different software projects. These multipliers and factors have rating levels based on the project under development. The rating levels of the effort multipliers and the scale factors are described in detail in Appendix “C”. The constant values (i.e. “A”= 2.94 and “B”= 0.91) are also calibrated from the existing software projects.

\[
PM = A \times (Size)^E \times \prod_{i=1}^{17} EM_i, \text{where } E = B + (0.1 \times \sum_{j=1}^{5} SF_j),
\]

\[A = 2.94, \text{and } B = 0.91 \quad \text{(COCOMOII. 2000)} \quad \text{(Eq.1)}\]

\[
TDEV = C \times (PM)^F, \text{where } F = D + 0.2 \times (E - B),
\]

\[C = 3.67, \text{and } D = 0.28 \quad \text{(COCOMOII. 2000)} \quad \text{(Eq.2)}\]
Second, the time needed for the development of a software system (TDEV) can be calculated using Equation 2. The time is calculated based on the development effort in person months (PM) with respect to the five scale factors that are used to calculate the parameter “F”, and some constant values which are “C”= 3.67 and “D”= 0.28.

To calculate the effort and the time for a system’s maintenance, the post-architecture model can be used. The maintenance is the process of modifying an existing software for updating and repairing purposes [224]. This process includes re-designing the software system, and re-coding parts of the system. Therefore, the evolution process (described in this thesis) corresponds to the maintenance process. Thus, we use the COCOMO II to calculate the effort required for evolving a context-aware adaptive system. To compute the evolution (maintenance) effort, Equation 1 is used. However, the number of efforts multipliers is only fifteen as the two multipliers: “required development schedule” and “required reusability” are not needed. In addition, there are different rating levels for the effort multiplier “required reliability” (see Appendix “C” for these rating levels). On the other hand, to estimate the time needed for the system maintenance (evolution), Equation 2 is used without any modifications.

To take into account the effect of modern software tools in calculating the development and maintenance efforts of a system, an extension to the effort multiplier “use of software tools” (TOOLS) is introduced [226]. The extension aims to show the critical role that modern software tools play in reducing the engineering effort, where three effort multipliers (i.e. completeness of tool coverage, tool integration, and tool maturity) are used for calculating the multiplier “use of software tools”. The “completeness of tool coverage” factor (TCOV) defines to what extent a tool supports the different activities that need to be performed in order to develop and evolve a software system. The “tool integration” factor (TINT) specifies to what degree the different tools that are used for developing and evolving a system can be integrated with each other. The “tool maturity” factor (TMAT) specifies for how long a tool is available and the technical support it provides. The rating levels for the three factors are presented in Appendix “C”. Using the three factors, the “use of software tool” multiplier can be calculate by Equation 3 [226].

\[
TOOLS = 0.51 \times TCOV + 0.27 \times TINT + 0.22 \times TMAT 
\]  \hspace{1cm} (Eq.3)

The post-architecture model is also used for estimating the number of staff needed for a project and the project cost [224]. First, the number of staff is calculated as the ratio between the effort and the time to develop a software system (i.e. PM/TDEV). For example, if the PM of a project is “30” and the TDEV equals to “10”. Then, the project can be completed by three persons (i.e. 30/10) in ten months. Second, the project cost is calculated as the expenses for the
project staff during the project period. For example, if a person costs the company $5K/month (i.e. his salary and other expenses), then the above project costs 5 (average expenses per person) *3 (number of persons) *10 (project duration) which is $150K.

9.3.2.2 Effort Reduction in Developing and Evolving the Travel Guide System

Using COCOMO II, in this section, we estimate the effort, time, and cost that are required to develop and evolve the travel guide software system. Based on the characteristics of the travel guide system, we have selected suitable ratings for the effort multipliers (i.e. product, platform, personal, and project factors), and the project scale factors as shown in Table 9-7. These ratings are derived from the rating levels of the factors that are presented in Appendix “C”.

To estimate the effort reduction in developing the travel guide system, we first estimate the development effort that is needed when our approach is used. Then, the required effort when a set of traditional approaches are used (e.g. the UML class diagram for designing the system, and Java for implementing the system) is estimated. Finally, the difference between the two efforts is calculated. Our approach (as discussed in Section 9.3.1) reduces the development effort by assisting the software engineer in specifying, validating, designing, and realizing the software system. As such, the ratings for the effort multipliers that are related to these tasks are the main differences between our approach and existing ones. These multipliers include “developed for reusability”, “analyst capability”, and “use of software tools” (highlighted in Table 9-7). First, in our approach, the system implementation is generated from the system’s design model. Also, elements which are reusable in different systems are already develop and used in the generation process. Thus, the “developed for reusability” factor is nominal in our approach (i.e. “1”), while it is extra high (i.e. “1.24”) when a traditional approach (e.g. Java) is used where a number of elements need to developed and reused for developing different software systems in the future (e.g. developing a set of components to enable a system’s runtime adaptation).

Second, with our approach, a number of techniques are introduced to automate a set of engineering tasks (e.g. requirements validation and synthesis of a system’s design model). This automation reduces the manual effort required from the analyst team (i.e. persons who work on specifying the system requirements and designing the system). Thus, the “analyst capability” factor is very high (i.e. “0.71”) when our approach is used, while it is nominal otherwise (i.e. “1”) as shown in Table 9-7.

Third, we have developed a tool (described in Chapter 7) that implements the techniques we have introduced to assist the software engineer in developing context-aware adaptive systems. Thus, the rating level for our tool is “0.92” where the tool covers the different engineering tasks,
integrates these engineering tasks together, and is available more than a year ago. On the other hand, the rating level for a traditional approach is “1.02” where the existing tools do not fully support all engineering tasks, cannot be easily integrated where they are developed separately, and are available more than two years ago.

Table 9-7: Effort estimation for developing and evolving the travel guide software system

<table>
<thead>
<tr>
<th>Phase</th>
<th>Development</th>
<th>Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Which approach?</td>
<td>Our Approach</td>
</tr>
<tr>
<td>Product Factors</td>
<td>1.1 Software Reliability</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>1.2 Data Base Size</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>1.3 Product Complexity</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td>1.3.1 Control Operations</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>1.3.2 Computational Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.3.3 Device-dependent Operations</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>1.3.4 Data Management Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.3.5 User Interface Management</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>1.4 Developed for Reusability</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.5 Documentation</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>2.1 Execution Time Constraint</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>2.2 Main Storage Constraint</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2.3 Platform Volatility</td>
<td>1</td>
</tr>
<tr>
<td>Platform Factors</td>
<td>3.1 Analyst Capability</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>3.2 Programmer Capability</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>3.3 Personnel Continuity</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3.4 Applications Experience</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>3.5 Platform Experience</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>3.6 Language and Tool Experience</td>
<td>0.91</td>
</tr>
<tr>
<td>Personal Factors</td>
<td>4.1 Use of Software Tools</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>4.1.1 Tool Coverage</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>4.1.2 Degree of Tool Integration</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>4.1.3 Tool Maturity</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4.2 Multi-site Development</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>4.3 Development Schedule</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1. Precededness</td>
<td>1.24</td>
</tr>
<tr>
<td></td>
<td>2. Development Flexibility</td>
<td>2.03</td>
</tr>
<tr>
<td></td>
<td>4. Team Cohesion</td>
<td>2.19</td>
</tr>
<tr>
<td>Scale Factors</td>
<td>5. Process Maturity</td>
<td>4.68</td>
</tr>
<tr>
<td></td>
<td>5. Project Size</td>
<td>3.5 KLOC</td>
</tr>
<tr>
<td></td>
<td>Average Person Cost</td>
<td>$ 5K</td>
</tr>
<tr>
<td>Other</td>
<td>Effort (PM)</td>
<td>8.42</td>
</tr>
<tr>
<td></td>
<td>Time (TDEV)</td>
<td>7.08</td>
</tr>
<tr>
<td></td>
<td>Number of Persons</td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>Total Cost</td>
<td>$ 42.08 K</td>
</tr>
</tbody>
</table>

Using the factors’ ratings specified in Table 9-7, the efforts for developing the travel guide software system with and without our approach are estimated. The effort in person months using
our approach is “8.42” while it is “16.32” when our approach is not used. Thus, our approach reduces the effort by “7.90” person months. In addition, the time for developing the travel guide system using our approach is “7.08” months when the number of staff is around “1.2”, while the time is “8.69” months when around “1.9” persons develop the system by a traditional approach (see Table 9-7). Furthermore, the overall project cost when our approach is used is “$42.08K”, while it is “$81.62 k” otherwise (i.e. the cost reduction is “$39.54K”). These results show that our approach is able to reduce the effort, time, and cost that are needed for developing the travel guide software system.

In the same manner, we have estimated the reduction in the engineering effort that is needed for evolving the system to cope with unanticipated changes that are discussed in Chapter 6. The differences between the factors’ ratings which are used to estimate the development effort and the factors’ ratings for estimating the evolution effort are: (1) the effort multipliers “developed for reusability” and “development schedule” are not used, and then we have removed them; (2) there are different rating levels for the multiplier “required system reliability”, and then new ratings are used as shown in Table 9-7. Using COCOMO II, the effort to evolve the travel guide system in person months is “0.97” when our approach is used. In the case of our approach is not used, the required effort is “1.52” (i.e. the effort reduction is “0.55” person months). In addition, the time to evolve the system by our approach is “3.64” months when the staff number is around “0.27” persons, while “0.36” persons work for “4.18” months to do that task using a traditional approach. Finally, the cost to evolve the system is “$4.87K” when our approach is used, while it is “$7.62K” otherwise as shown in Table 9-7. Such results demonstrate that our approach is able to evolve the system with less effort, cost, and time than a traditional approach.

The COCOMO II is a widely accepted model and is calibrated using a number of software projects. However, there still a risk and a degree of uncertainty because of the parametric inputs of the model [227]. To cope with this risk and uncertainty, we use the Monte Carlo simulation method [228-229]. This method enables the estimation of the average effort, time, and cost for the development and evolution a context-aware adaptive software system.

Following the Monte Carlo simulation, a random number generator is used to specify input ranges for the parameters of COCOMO II. These ranges are derived based on the actual values of the model parameters (i.e. the effort multipliers and scale factors described in Appendix “C”). Then, these ranges are used for estimating the effort, time, and cost of developing or evolving a software system. First, to generate a range for an input parameter, a random number between “0” and “1” is generated. Then, a value of “0.5” is subtracted from the generated values, so that the result is a set of positive and negative values. After that, these values are multiplied by the
varying range of an input parameter and added to its actual value to generate randomized values for the input parameter. For example, in the case of the value of an input parameter is “1.1” and its varying range is “0.06”, we generate a set of random numbers such as “0.47, 0.12, 0.57, and 0.62”. Then, we subtract “0.5” from them to get the values for manipulating the varying range which are “-0.03, -0.38, 0.07, and 0.12”. Finally, the varying range “0.06” is multiplied by these values and added to the actual value “1.1”, and then the generated range for the actual value becomes “1.098, 1.077, 1.104, and 1.107”. Following this method, ranges for all parameters of the COCOMO II are generated. Second, the generated ranges of the parameters are fed to the equations (described above) to estimate the effort, time, and cost for developing and evolving a context-aware adaptive software system.

Using the Monte Carlo simulation, we have generated ranges (with size of 50) for the input parameters of the COCOMO II to estimate effort, time, and cost for developing and evolving the travel guide system with and without our approach. First, the top of Figure 9-1 shows the estimated effort when our approach is used. The average (mean) value for the development effort in person months is “8.74”. The bottom of Figure 9-1 represents the estimated effort when a traditional approach is used. In that case the average estimated effort is “16.06”. Therefore, the effort reduction gained by our approach is 45.6% (=(16.06-8.74)/16.06)*100). The time and the cost for developing the travel guide system depend on the estimated effort. As such, the graphs that represent the estimated time and cost using the Monte Carlo simulation have the same variations of the estimated effort shown in Figure 9-1 (see Appendix “D” for the details).

Figure 9-1: The estimated effort for developing the travel guide system using the Monte Carlo simulation
Second, the estimated effort to evolve the travel guide system in response to unanticipated changes is shown in Figure 9-2. The top of Figure 9-2 shows the required effort to evolve the system using our approach with an average of “0.98”, while the bottom of Figure 9-2 represents the estimated effort when our approach is not used and it is average is “1.52”. Consequently, the reduction in the effort required to evolve the travel guide system is “35.4%”. The estimated cost and time for evolving the travel guide software system (that are similar to the estimated effort) are discussed in detail in Appendix “D”.

Table 9-8 shows a summary for the average effort, time, and cost that are needed for the development and evolution of the travel guide system. It also shows the obtained reduction in these aspects when our approach is used relative to using a traditional approach. In order to measure the reduction in the development time, we have made the number of staff in the project fixed. This number is “5” during the system development, and “2” at the system evolution stage. Therefore, we calculate the reduction by the difference between the times that this number of persons can take to develop or to evolve the travel guide system (see Table 9-8). In general, the effort, time, and cost for developing the travel guide system are reduced by around 45%, while the reduction is around 35% in evolving the system as shown in Table 9-8.
Table 9-8: Summary of the average effort reduction in developing and evolving the travel guide system

<table>
<thead>
<tr>
<th>Phase</th>
<th>Development</th>
<th>Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which approach?</td>
<td>Our Approach</td>
<td>Traditional</td>
</tr>
<tr>
<td>Effort (PM)</td>
<td>8.74</td>
<td>16.06</td>
</tr>
<tr>
<td>Time (TDEV)</td>
<td>1.75</td>
<td>3.21</td>
</tr>
<tr>
<td>Number of Persons</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Total Cost</td>
<td>$43.93 K</td>
<td>$80.33 K</td>
</tr>
</tbody>
</table>

9.3.2.3 Effort Reduction in Developing and Evolving the Electronic Exam System

We also have used COCOMO II for estimating the effort, time, and cost that are required to develop and evolve the electronic exam system. Based on the e-exam system’s characteristics, ratings for the effort multipliers and the scale factors are selected as shown in Table 9-9. The differences between the factors’ ratings when our approach is used and a traditional approach is used are the ratings for the two multipliers: “analyst capability” and “use of software tools” as discussed in the previous section. However, the multiplier “developed for reusability” has the same rating in the two cases (see Table 9-9), because the development team have considered this factor during the development of the travel guide system (see above) and there is no need to develop any other reusable components.

Similar to the above, we have used the factors’ ratings specified in Table 9-9 to estimate the efforts for developing the e-exam system with and without our approach. The development effort of the system by our approach is “27.61”, while it is “43.18” when a traditional approach is used (i.e. the effort reduction equals to “15.57”). In addition, the time required to develop the system using our approach is “10.45” months when the number of persons in the development team is around “2.64”, while the time is “12.03” months when the staff is round “3.59” persons and a traditional approach is used. Finally, the project’s overall cost is “$ 138.03K” with our approach and “$ 215.9 k” otherwise (see Table 9-9). These results demonstrate that the effort, time, and cost that are needed for developing the e-exam software system are reduced when our approach is used.

We have also estimated the efforts that are needed for the e-exam system’s evolution to cope with unanticipated changes (described in Chapter 8). Using COCOMO II, the effort needed to evolve the system is “3.16” person months when our approach is used, while the effort is “4.95” when a traditional approach is used (see Table 9-9). In addition, around “0.6” persons need to work for “5.28” months to evolve the system by our approach, while “0.81” persons work for “6.08” months to perform the evolution task without our approach. Finally, evolving the system in response to the unanticipated changes costs “$ 15.82K” when our approach is used, and the
cost is “$ 24.74K” otherwise as shown in Table 9-9. As such, our approach is able to evolve the e-exam system with less effort, cost, and time compared to using a traditional approach.

Table 9-9: Effort estimation for the development and evolution of the e-exam system

<table>
<thead>
<tr>
<th>Phase</th>
<th>Development</th>
<th>Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Our Approach</td>
<td>Traditional</td>
</tr>
<tr>
<td>Product Factors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1 Software Reliability</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>1.2 Data Base Size</td>
<td>1.28</td>
<td>1.28</td>
</tr>
<tr>
<td>1.3 Product Complexity</td>
<td>1.57</td>
<td>1.57</td>
</tr>
<tr>
<td>1.3.1 Control Operations</td>
<td>1.17</td>
<td>1.17</td>
</tr>
<tr>
<td>1.3.2 Computational Operations</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.3.3 Device-dependent Operations</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.3.4 Data Management Operations</td>
<td>1.34</td>
<td>1.34</td>
</tr>
<tr>
<td>1.3.5 User Interface Management</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.4 Developed for Reusability</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.5 Documentation</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Platform Factors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 Execution Time Constraint</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2.2 Main Storage Constraint</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2.3 Platform Volatility</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td>Personal Factors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1 Analyst Capability</td>
<td>0.71</td>
<td>1</td>
</tr>
<tr>
<td>3.2 Programmer Capability</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>3.3 Personnel Continuity</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.4 Applications Experience</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>3.5 Platform Experience</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>3.6 Language and Tool Experience</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>Project Factors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1 Use of Software Tools</td>
<td>0.92</td>
<td>1.02</td>
</tr>
<tr>
<td>4.1.1 Tool Coverage</td>
<td>0.9</td>
<td>1.09</td>
</tr>
<tr>
<td>4.1.2 Degree of Tool Integration</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td>4.1.3 Tool Maturity</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>4.2 Multi-site Development</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>4.3 Development Schedule</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Scale Factors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Precedentedness</td>
<td>2.48</td>
<td>2.48</td>
</tr>
<tr>
<td>2. Development Flexibility</td>
<td>4.05</td>
<td>4.05</td>
</tr>
<tr>
<td>4. Team Cohesion</td>
<td>2.19</td>
<td>2.19</td>
</tr>
<tr>
<td>5. Process Maturity</td>
<td>4.68</td>
<td>4.68</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Size</td>
<td>10 KLOC</td>
<td>10 KLOC</td>
</tr>
<tr>
<td>Average Person Cost</td>
<td>$ 5K</td>
<td>$ 5K</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effort (PM)</td>
<td>27.61</td>
<td>43.18</td>
</tr>
<tr>
<td>Time (TDEV)</td>
<td>10.45</td>
<td>12.03</td>
</tr>
<tr>
<td>Number of Persons</td>
<td>2.64</td>
<td>3.59</td>
</tr>
<tr>
<td>Total Cost</td>
<td>$ 138.03 K</td>
<td>$ 215.90 K</td>
</tr>
</tbody>
</table>

To cope with the risk and uncertainty because of the parametric inputs of COCOMO II, we have used the Monte Carlo simulation to estimate the average effort, time, and cost that are needed for developing and evolving the e-exam software system. First, in Figure 9-3, we show the estimated effort for developing the e-exam system. The estimated effort in person months
using our approach is shown in the top part of Figure 9-3, where the average effort is “27.87”. The bottom part of Figure 9-3 represents the estimated effort when our approach is not used and the average estimated effort is “43.00” in that case. As such, the reduction in the development effort that is obtained by our approach is 35.2%. Compared to the effort reduction gained in the travel guide system example (i.e. 45.6%), this reduction percentage is lower because in the first example the development team has developed a set of reusable components. These components enable the realization of the adaptability aspect of the e-exam system, and then the development effort of this system is reduced. The time and the cost for developing the e-exam system depend on the estimated effort. Thus, the graphs that represent the estimation for the time and cost for developing the e-exam system using the Monte Carlo simulation have the same variations of the estimated effort presented in Figure 9-3 (further details are in Appendix “D”).

![Figure 9-3: The estimated effort for developing the e-exam system using the Monte Carlo simulation](image)

Second, in response to unanticipated changes (discussed in Chapter 8), the e-exam system needs to evolve. The effort estimation for evolving the e-exam system is shown in Figure 9-4. In the top of Figure 9-4, the required effort to evolve the system by our approach with an average of “3.17” is represented. In addition, the bottom of Figure 9-4 shows the estimated effort when a traditional approach is used, and its average is “4.99”. Consequently, the reduction in the effort
required to evolve the e-exam system is 36.4% of the total effort. The average time and cost to evolve the system are described in the Appendix “D”, where they are similar to the reduction in the system’s evolution effort.

Figure 9-4: The estimated effort for evolving the e-exam system using the Monte Carlo simulation

Table 9-10 shows a summary of the estimated average effort, cost, and time for developing and evolving the e-exam system. It also shows the gained reductions in these aspects when our approach is used compared to not using our approach. In general, the effort, time, and cost for developing the e-exam system are reduced by around 35%, while the effort reduction is around 36% in evolving the system as shown in Table 9-10.

Table 9-10: The average effort, time, and cost for developing and evolving the e-exam system

<table>
<thead>
<tr>
<th>Phase</th>
<th>Development</th>
<th>Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort (PM)</td>
<td>Our Approach 27.87</td>
<td>Traditional 43.00</td>
</tr>
<tr>
<td></td>
<td>Our Approach 3.17</td>
<td>Traditional 4.99</td>
</tr>
<tr>
<td>Time (TDEV)</td>
<td>5.57</td>
<td>8.60</td>
</tr>
<tr>
<td></td>
<td>1.59</td>
<td>2.50</td>
</tr>
<tr>
<td>Number of Persons</td>
<td>5 Fixed</td>
<td>5 Fixed</td>
</tr>
<tr>
<td>Total Cost</td>
<td>$ 139.47 K</td>
<td>$ 214.96 K</td>
</tr>
<tr>
<td></td>
<td>$ 15.88 K</td>
<td>$ 24.99K</td>
</tr>
</tbody>
</table>
9.4 Performance Evaluation

Adding the adaptability feature to a software system comes with a cost which is the CPU time needed to perform the runtime adaptation in response to context changes. As such, this runtime cost needs to be quantified. In addition, we have introduced a number of algorithms to automate a set of engineering tasks in the development and evolution of context-aware adaptive systems. Therefore, the time complexity of these algorithms needs to be evaluated to ensure that these algorithms are usable in large scale software systems.

9.4.1 The Time Required for Adapting a System at Runtime

To enable the runtime adaptation of a software system, the system needs to have a management composite that adapts the system functionality in response to context changes. This adaptability feature comes with a runtime cost [64, 230]. In the following, we present the evaluation of this runtime cost by measuring the time required for performing the system’s runtime adaptation in response to context changes. A PC with Intel Core 2 Due 3 GHZ CPU and 3 GB RAM is used as the test-bed.

The time needed for adapting a system at runtime can be calculated by the time required for monitoring the context changes, deciding the required adaptation actions, and acting the actions.

(1) Monitoring the Context. In our approach, there is a need to keep track of some context attributes that trigger the system adaptation. When any of the attributes that the adaptation rules are interested in changes, this change is notified to the organizer player to decide the needed adaptation. The time required to notify the organizer player of a context attribute change equals to “14.59” milliseconds on average.

(2) Deciding the Required Adaptation Actions. When the context is changed, the adaptation rules need to be evaluated to decide the required adaptation actions. The time required by the decision making process is laid in the rules loading time at the beginning and their execution time in response to context changes. To measure this time, we have used sets of rules with sizes “10, 20, 30, 40, and 50”. Figure 9-5.A shows that the rules loading time varies from “1.81” to “2.05” seconds based on the adaptation rules size. This is not an overhead where it is performed once at the system start-up. In addition, the time required for executing the adaptation rules is between “14.8” to “29.9” milliseconds (see Figure 9-5.B), which cannot be considered as a long time to decide the adaptation actions. Generally, the rules loading and execution times increase in a linear manner with the number of rules as shown in Figure 9-5, which means the approach is useable in large scale software systems.
Figure 9-5: The adaptation rules’ loading and execution times

(3) Acting the Adaptation Actions. Different adaptation actions can be performed to adapt a system in response to context changes. Table 9-11 summarises the average time needed to apply the major adaptation actions in milliseconds. It shows the actions for adding and removing some of the system elements, where they are of interest from the user point of view (i.e. the elements are added or removed in response to changes in the user needs). These actions are performed in a short time while the system in operation. As such, we can conclude that they do not affect the user interactions with the system.

Table 9-11: The time (in milliseconds) required to apply the adaptation actions

<table>
<thead>
<tr>
<th>Adaptation Action</th>
<th>Required Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Functional/Context Role</td>
<td>198.953(^{19})</td>
</tr>
<tr>
<td>Remove Functional/Context Role</td>
<td>0.207</td>
</tr>
<tr>
<td>Add Functional Contract</td>
<td>0.204</td>
</tr>
<tr>
<td>Remove Functional Contract</td>
<td>1.129</td>
</tr>
<tr>
<td>Add Contextual Contract</td>
<td>1.565</td>
</tr>
<tr>
<td>Remove Contextual Contract</td>
<td>0.052</td>
</tr>
<tr>
<td>Change Role-Player Binding</td>
<td>0.035</td>
</tr>
<tr>
<td>Add Adaptation Rule</td>
<td>0.096</td>
</tr>
<tr>
<td>Remove Adaptation Rule</td>
<td>0.918</td>
</tr>
</tbody>
</table>

An example of a delay that the tourist can experience in using the travel guide system occurs when he wants to include the route planning service. To include this service, it takes around 422 milliseconds which cannot be considered as a delay of significance to the tourist.

\(^{19}\) Adding a role takes a long time, where a Java class corresponding to that role need to be created. Then, this class is complied and deployed to the Apache Axis2 (http://axis.apache.org/axis2/java/core/).
9.4.2 The Time Complexity of the Algorithms

In our approach, we have introduced a set of algorithms to automate a number of engineering tasks, i.e., validating a system’s requirements, synthesizing a system’s design model from its scenarios, updating a system’s design model to reflect the changes in the system scenarios, and computing differences between a running system model and its evolved model. We analyse the time complexity of these algorithms below to ascertain that they are usable (work effectively) in the development and runtime evolution of large scale context-aware adaptive systems.

(1) The Requirements Validation Algorithm: The algorithm we have introduced in Section 4.2 to validate a system’s requirements depends on the number of adaptation conditions that the system have. As such, the larger the number of adaptation conditions, the longer the time the algorithm will take to execute. The time complexity\(^{20}\) of this algorithm is \(O(M \times N)\), where “M” is the number of the adaptation scenarios in the system while “N” is the number of adaptation fragments (conditions) in these scenarios.

(2) The Algorithm to Synthesize a System’s Design Model from its Scenarios: In Section 5.3, we have introduced a number of algorithms for synthesizing a system’s design model from its scenarios’ descriptions. In general, the algorithms parse the system’s scenarios (functional or adaptation), and then for each scenario element a corresponding element in the system design model is generated. Each algorithm has two loops: the first loop iterates over the scenarios (e.g. “M” scenarios), and the second loop parses a scenario to generate the elements corresponding to the scenario’s elements (e.g. “N” elements). Therefore, the time complexity of these algorithms is \(O(M \times N)\).

(3) The Algorithm to Select and Apply an Evolution Pattern to a System’s Model: To specify the changes to a system’s design model in response to changes in the system scenarios, we have introduced an algorithm in Section 6.2.2. The algorithm is a switch-statement that selects a proper change pattern to be applied into the system model when a change is performed on the system scenarios, and then the time complexity of this algorithm is \(O(1)\).

(4) The Algorithm to Compute a Script to Evolve a Running Software System: The algorithm for indentifying the differences between a running instance of the system and its evolved model has been introduced in Section 6.3. The algorithm looks for elements that have been removed, added, or modified in the running system’s model. The added elements are identified by getting an element (i.e. Loop 1) from the new model and compare this element with each element in the

---

\(^{20}\) This calculation does not include the time required for performing the validation of the system variants using the Romeo tool.
model of the running system (i.e. Loop 2). When there is no match found, it means the element is new and need to be added to the system by adding a set of adaptation actions to an evolution script. A similar mechanism is used for identifying the modified and removed elements. Thus, the computational complexity of the algorithm for generating the evolution script is $O(M \times N)$, where “$M$” is the number of elements in the new (evolved) model of the system while “$N$” is the number of elements in the running system’s model.

**9.5 Summary**

In this chapter, we have evaluated the benefits of our approach by carrying out four types of evaluations. First, we have performed a feature-based analysis of the approach. In this analysis, we have listed the features that an approach for developing and evolving context-aware adaptive systems should support, and then for each feature we have discussed how it is supported in our approach systematically. We have also compared our approach with existing approaches relative to these features. The results of the feature-based analysis show that our approach fully supports most of the features that need to be considered during the development and runtime evolution of context-aware adaptive systems. However, there are a small number of features that are partially supported by the approach. Thus, a number of enhancements will be performed to our approach to fully support these features in the future.

Second, to evaluate the effectiveness of the visual notations we have introduced to specify the system requirements and to enable the system design, we have used the physics of notations theory. We first described the nine evidence-based principles of the theory that are introduced to evaluate a modelling notation. Then, we used the principles to evaluate our graphical notations. The evaluation result shows that our notations have fully considered six of the nine principles, while the other three principles have been considered partially. With respect to the partially considered principles, the graphical notations will be improved to fully consider such principles in the future.

Third, to assess the effort reduction obtained when our approach is used for developing and evolving context-aware adaptive software systems, we have used the constructive cost model (COCOMO) II. We first used this model to estimate the efforts needed to develop and evolve a software system with and without our approach. The effort reduction is then calculated as the difference between these efforts. To reduce the risk and uncertainty because of the parametric inputs of the COCOMO model, we have used Monte Carlo simulation to estimate the average effort, time, and cost that are needed to develop and evolve context-aware adaptive software systems. In general, our approach is able to reduce the effort, time, and cost that are required for
the development and runtime evolution of context-aware adaptive systems. In the travel guide system example, the reduction in the development effort is about “45%”, while the reduction in the evolution effort is around “35%”. In the e-exam system example, the development effort is reduced by around “35%”, while the reduction of the system’s evolution effort is about “36%”. The reduction in the development effort of the e-exam system is less than the reduction in the development effort of the travel guide system because a set of components to realize the system adaptability have been developed with the first example and re-used in the second example (i.e. in the e-exam system).

Fourth, performance evaluations have been performed to assess the runtime cost of adding the runtime adaptability feature to a software system, and to quantify the time complexity of the algorithms we have introduced to automate some of the engineering tasks in our approach. The results of the evaluations show that: (1) the cost of adding the adaptability feature to a system is small and does not affect the user interactions with the system; (2) the time complexity of the algorithms we have introduced varies from O (1) to O (M*N), and then these algorithms can be used for developing and evolving large scale context-aware adaptive systems.
Chapter 10: Conclusion

In this thesis, we have presented an approach to assist the software engineer in the development and runtime evolution of context-aware adaptive software systems. Our approach is scenario-driven where a system’s requirements are captured as two sets of scenarios (i.e. functional and adaptation). These scenarios are then validated by checking the consistency of the adaptation scenarios, and the conformance of the functional scenarios to the system properties. In addition, the system’s design model is automatically synthesized from the system scenarios. This model can be translated to an executable ROAD model to enable the system realization. Furthermore, we enable the system runtime evolution by changing the system scenarios, and then the changes to the scenarios are reflected to the system’s design model and its realization automatically. The motivation of this approach stems from the need to address limitations of existing approaches to developing and evolving context-aware adaptive software systems. The outcome of this study is a novel scenario-driven approach to assist the software engineer in the development and runtime evolution of context-aware adaptive software systems.

In this chapter, we first highlight the main contributions of the thesis to the field of context-aware adaptive software systems. Then, we discuss future work to further improve our approach to developing and evolving context-aware adaptive software systems.

10.1 Contributions

In recent years, a number of approaches have been introduced to assist the software engineer in performing a set of engineering tasks to develop and evolve context-aware adaptive software systems. However, these approaches have a number of limitations.

First, approaches that have been introduced to specify a system’s requirements capture the system requirements as a set of goals. But, it is often difficult for the stakeholders to articulate their needs as goals [22, 37]. In addition, in existing approaches, the adaptation requirements are intertwined with the functional requirements specification. Furthermore, they do not explicitly represent context information and its relationships with the system’s functional and adaptation requirements. Therefore, it is difficult to use these approaches for specifying the requirements of a context-aware adaptive software system.

Second, to validate a system’s functional requirements using existing approaches, the system variants and the properties to be checked against these variants need to be enumerated which are a large number in large scale systems. In addition, the existing approaches do not fully support
the validation of the system’s adaptation requirements, where they do not check the consistency of adaptation actions that need to be applied into the system in response to context changes. As such, the existing approaches are not suitable for validating the requirements of a context-aware adaptive software system.

Third, to design a context-aware adaptive system using the existing approaches, the software engineer is responsible for translating the system requirements to a design model manually. As such, the causal connection between the system design model and its requirements is maintained manually which is a difficult task in complex systems. In addition, little attention has been paid to design the system’s structure and behaviour in a coherent manner. Furthermore, the system’s three aspects (i.e. functionality, context, and adaptive behaviour), and their relationships are not explicitly represented. Therefore, the system aspects and their relationships cannot be clearly captured and easily adapted (manipulated) to cope with runtime context changes.

Fourth, limited attention was given to support runtime evolution of a context-aware adaptive system in response to unanticipated changes. Existing approaches enable the software engineer to evolve the system’s requirements specification and its design model in an ad-hoc manner, where there is not a guideline or a set of patterns to support the engineer in performing that task. In addition, to validate the changed requirements specification, the whole validation process that was performed at the development time needs to be performed again which is inefficient and a time consuming task. Furthermore, to realize the system’s design changes to a running system, there are a set of code-based and script-based approaches. But, using these approaches to realize the system changes is tedious and error prone, where the runtime changes need to be performed (specified) at a lower level of abstraction.

To tackle these limitations, in this thesis, we have introduced a scenario-driven approach to assist the software engineer in the development and runtime evolution of context-aware adaptive software systems. The approach makes four main contributions:

(1) **Requirements Specification:** Our approach specifies a system’s requirements as two sets of scenarios: functional and adaptation. The functional scenarios represent the system’s core functionality while the adaptation scenarios specify the system adaptation in response to context changes.Specifying the system requirements by this approach allows the stakeholders to specify their needs as a set of scenarios which is easier than specifying such needs as a set of goals [22, 231]. In addition, we specify the functional and adaptation requirements of the system into two separate groups of scenarios, so that the system requirements can be clearly captured and easily managed. Furthermore, we represent the context information explicitly in these scenarios. Thus,
the context information and its relationships with the system’s functionality and adaptation are clearly represented.

(2) Requirements Validation: To validate a system’s requirements, we specify the system’s properties that need to hold while the system in operation in a form similar to the scenarios. In addition, our approach automatically generates a large number of system variants from the system scenarios and identifies the properties that need to be checked against these variants. The variants and the system properties are then transformed to formal models to enable the variants’ validation. As such, the task of validating the system’s functional requirements becomes easier. Furthermore, we support the validation of the system’s adaptation requirements by generating different adaptation scripts, which can be applied to the system in response to context changes. The generated scripts are then automatically checked to ensure the consistency of the system’s adaptation requirements.

(3) The System’s Design and Realization: To ease the task of designing a context-aware adaptive system, we have introduced an organization-based meta-model. Following this meta-model, the system’s three aspects (i.e. functionality, context, and adaptive behaviour), and their relationships are clearly captured. In addition, the system’s structure and its behaviour and how they are related to each other are explicitly represented. The meta-model represents the system as two main composites: functional and management.

The functional composite specifies the system functionality. This composite has two parts: structure and behaviour. The functional structure captures the system’s functional elements as functional roles that are played by players. These roles are connected by a set of functional contracts to define their permissible interactions. The functional structure also has contextual roles bound to context providers to provide the context information that is needed by the system functionality. The system’s functional behaviour is modelled as a set of event-based processes. Each process consists of a set of tasks that are related to each other by a set of events.

The management composite includes a set of elements that are responsible for adapting the system at runtime. These elements include a set of context roles bound to context providers to provide context information that triggers the system adaptation. To decide the system reactions to cope with context changes, this composite has an organizer role bound to an organizer player. The composite also has a functional system role that represents the system’s functionality that needs to be adapted in response to context changes.

We also have introduced a set of algorithms that automatically derive the system’s design model from its requirements (specified as two sets of scenarios). Thus, the causal connection
between the system requirements and its design is maintained. To enable the system realization, we transform the system’s design model to an executable ROAD model. When this model is deployed to the ROAD runtime environment, an instance of the system is created and different adaptation actions can be applied to this instance to cope with runtime context changes.

(4) The System’s Runtime Evolution: To enable a system’s runtime evolution, first, we have introduced a set of change patterns. These patterns specify changes to the system’s requirements in response to unanticipated changes. Thus, the software engineer can easily specify changes to the system requirements following such patterns. Second, to validate the changed requirements specification, we have presented a technique that automatically identifies the system variants that need be validated and the properties to be checked against them. Then, these variants and their properties are transformed to formal models to enable the variants’ validation. Thus, only a small number of variants that are related to the changed requirements are validated. A similar technique is used for identifying adaptation scripts that need to be checked for consistency.

Third, to reflect changes in the system requirements to its design model, we have introduced an algorithm that derives changes to the system’s design model from the changed requirements (specified as scenarios). Fourth, to realize the changes of the system’s design model to a running instance of the system, we compute the differences between the running system’s model and the evolved system model. Then, an evolution script that contains adaptation actions corresponding to the differences is generated and executed to apply the adaptation actions to the system while in operation. Therefore, the software engineer specifies the changes to the system requirements only, and then the design changes are derived from the changed requirements and realized to the running system automatically.

Overall, our scenario-driven approach addresses the requirements that need to be considered in the development and runtime evolution of context-aware adaptive software systems. It assists the software engineer in specifying a context-aware adaptive system’s requirements, validating the requirements specification, designing and realizing the system, and evolving the system in response to unanticipated changes.

10.2 Future Work

In this thesis, we have introduced a scenario-driven approach to assist the software engineer in performing a set of engineering tasks to develop and evolve context-aware adaptive systems. However, there is still much work to be done to further enhance the approach. In the following, we list some possible future work. We classify the work into four groups.
Specifying the System Requirements

- **Uncertainty of Functional Requirements**: A context-aware adaptive system may be deployed into an environment that is not totally anticipated at the design time, and then the achievement of the system’s functional requirements is affected by such unanticipated changes. Thus, there is a need to relax the functional requirements to cope with such environment uncertainty at the development time [19, 54]. In our approach, we deal with the unanticipated changes by evolving the requirements specification to take into account these changes at runtime. But, our requirements specification approach can be extended to consider the environment uncertainty at the development time by introducing relax operators (e.g. as early as possible to) that are proposed by Whittle et al. [54] to the functional scenarios.

- **Non-functional Requirements**: In our approach, we specify a system’s non-functional requirements as constraints on the functional scenarios. But, only measurable quality requirements such as response time and availability can be represented. To capture other qualities such as security requirements, our approach needs to be extended.

- **Contextual Requirements**: To specify the context information that affects the system operation, we have introduced contextual lifelines. We also use contextual messages to specify context attributes that are required by the system, i.e., the context model is represented as a set of entities that have a set of context attributes. But, other aspects such as context attributes’ dependency, imperfection, etc. need to be considered [53, 232]. As such, our approach needs to be improved to capture such aspects.

- **Priority of Adaptation Actions**: In our approach, the set of adaptation actions that are generated from the adaptation scenarios to cope with context changes are performed based on the order of the adaptation fragments in the adaptation scenarios. But, some adaptation actions need to be performed quickly than other actions. Thus, priorities need to be defined for the adaptation actions, so that actions with higher priorities are applied first [38, 121]. To take the actions priorities into account, our adaptation scenarios need to be extended to specify such priorities. In addition, in generating an adaptation script, the actions with high priorities should be added to the script first.

Validating the Requirements Specification

- **Improving Functional Requirements Validation**: The system variants are generated from the same model, where there is a common base model for all variants, and there
are variations that are waved to the base model to form the variants. In recent years, a number of approaches have been introduced to validate software systems that have such characteristics (e.g. [233-235]). Thus, these approaches can be used to improve the performance of our approach to validating the functional requirements.

- **Validation of Adaptation Requirements**: In this thesis, we have focused on validating the adaptation scripts that are generated from the adaptation scenarios in response to context changes to ensure their consistency. But, the context information consistency needs to be also checked. In addition, there is a need to ensure that correct adaptation rules are triggered in response to the context changes. To consider these aspects, the approach proposed by Xu et al. [26] can be used to ensure the context consistency, and the technique introduced by Sama et al. [25] can be used to ascertain that correct adaptation rules are fired in response to the context changes.

**The System’s Design and Realization**

- **Improving the Organization-based Meta-model**: Similar to functional requirements specification, our design meta-model only represents measurable qualities such as response time. Thus, the meta-model needs to be improved to capture other qualities (e.g. security). In the same manner, an extension to the design meta-model is needed to consider additional context properties (e.g. context attributes’ dependency), and priorities for the adaptation actions.

- **Improving Design Synthesis Algorithms**: In our approach, the functional scenarios are transformed to a functional composite, while a management composite is derived from the adaptation scenarios. In a large scale software system, there will be a large number of scenarios, and then two large functional and management composites are derived from the scenarios. Therefore, the synthesis algorithms need to be improved to decompose the scenarios into groups that are transformed to small functional and management composites. Then, the small composites are grouped to construct larger composites that represent a high level (hierarchical) view of the system.

- **Structure Validation**: To validate a context-aware adaptive system’s design, we have used the approach introduced by the ROAD framework [140]. However, the ROAD framework only supports the validation of the system behaviour. As such, to validate the system structure, a formal representation of the system structure that follows the Alloy language [124, 236] needs to be generated. Then, the Alloy analyser [237] can be used for validating the system structure against its constraints.
• **Existing Monitors:** At runtime, the system needs to be adapted to cope with changes in its functional qualities (e.g. response time, availability, etc). These qualities have standard methods to be calculated. For example, the response time can be calculated by computing the differences between the time of sending a request and the time to get the response back. Therefore, our approach can be enhanced by providing a set of monitors that implement such standard methods. These monitors are ready for use by the system developer, so that the effort to complete the code required for having a fully functioning system is reduced.

• **Powerful Context Analyser:** In response to context changes, we evaluate a number of conditions to identify which adaptation rules to be fired. This is a simple analysis of the context changes, where we assume the context model is a set of simple attributes. However, as pointed out above, our context model will be improved to consider the context dependency, imperfection, etc. As such, a more powerful analysis technique (e.g. [238-239]) needs to be used to consider such aspects.

• **Intelligent Decision Maker:** Condition-action rules are a simple approach to specify a system’s adaptive behaviour, and this adaptive behaviour is deterministic that can be easily validated. However, it only reacts to pre-specified context changes, and it cannot take any action when an unanticipated context situation occurs and this may lead to a system failure (i.e. system goals are not achieved) [123]. To address this problem, an approach that can automatically generate a set of actions that make the system re-achieve its goals at runtime needs to be used (e.g. [240-241]).

• **Safe Adaptation:** To ensure that the system is working properly, the system needs to be safely adapted by preserving a number of general constraints. In our approach, we have relied on the ROAD framework in doing that task. However, there are system specific states that need to be transferred after the adaptation [242-243] which are not considered in the ROAD framework. Thus, our approach needs to be extended to realize the transfer of the system’s specific states when the system is adapted to cope with runtime context changes.

**The System’s Runtime Evolution**

• **Identifying Other Evolution Patterns:** In this thesis, we have identified a number of change patterns that specify the changes to a system’s requirements specification and its design model to cope with unanticipated changes. A further investigation needs to be performed to identify other useful patterns.
• **User Driven Evolution:** To evolve a running system using our approach, the software engineer should perform that task. But, in some cases the communications between the engineer and the system are infrequent and sometimes impossible. Therefore, our approach needs to be improved to enable the end users to do some system evolutions by raising the changes patterns to the users’ level [244-245], so that the users can use the patterns easily. In addition, the approach needs to be extended to make the users able to select some of the existing services and integrate them with the system.

• **Validating the System’s Evolved Design Model:** The ROAD framework supports the validation of the system behaviour at the design time. However, it does not provide a technique to validate the evolved system’s structure and behaviour at runtime. Thus, a technique to validate the evolved system’s design model needs to be introduced.

• **The Functional Players and Context Providers Evolution:** Our approach focuses on evolving a system’s internal structure and behaviour with less attention to evolving the system’s functional players and context providers, where they are evolved by the system developers manually. Thus, our approach needs to be improved to enable the automatic (or semi-automatic) runtime evolution of the system’s functional players and context providers.
Appendices
Appendix A: The Travel Guide System’s Scenarios, Properties, and ROAD Model

In the thesis, examples from the travel guide system are provided to help the discussion of the approach. However, these examples do not describe some aspects of the travel guide system in a comprehensive manner. Therefore, we provide below comprehensive descriptions for the travel guide system’s scenarios and properties, and its deployable ROAD model. The other aspects of the travel guide system (e.g. the system design model) are described in a comprehensive manner in the thesis, and then we do not describe them below.

1. The Travel Guide System’s Scenarios and Properties

In Chapter 4, we introduced some scenarios and properties for the travel guide system. We also introduced some modifications to such scenarios and properties in Chapter 6. But, the full set of scenarios and properties of the travel guide system is not presented. As such, we describe below the system’s scenarios and properties in a comprehensive manner. Some of these scenarios and properties are replication of scenarios and properties presented in Chapters 4 and 6. The travel guide system consists of 6 functional scenarios, 6 adaptation scenarios, and 31 properties.

1.1 The Travel Guide System’s Functional Scenarios

The travel guide system includes 6 functional scenarios for specifying the user interactions with the overall system (FS1), finding attractions (FS2), planning a route (FS3), finding a restaurant (FS4), notifying the user with the vehicle speed and alerting him if needed (FS5), and updating the calculated routes in response to changes in the traffic information (FS6).

Functional Scenario 1 (Using the Travel Guide System): To allow the user interactions with the travel guide system, the scenario “FS1” is introduced (see Figure 1.A). In this scenario, the user is able to login to the system. After a successful login, he can use the system services (i.e. the attractions finder, the route planner, and the restaurant locator), and be alerted when the vehicle speed exceeds the speed limit.

Functional Scenario 3 (Plan a Route): To plan a route, the functional scenario “FS3” is used. This scenario starts with a user request to plan a route from a location to a destination as shown in Figure 1.B. Then, the route planner acquires the traffic information, the user’s attractions, and the user’s driving preferences from their providers. In the case that there is a set of attractions selected by the user, the “ProvideRoutes1” interaction is executed while the “ProvideRoutes2” interaction is used otherwise. Finally, the user can select a suitable route and start his journey.
This route is updated during the journey to take into account changes in the traffic information as shown at the end of Figure 1.B.

**Figure 1:** Functional scenarios for using the travel guide system and planning a route

*Functional Scenario 2 (Find Attractions):* The scenario “FS2” shown in Figure 2.A is used for finding attractions. In response to a user request to find attractions (i.e. “FM1”), the weather condition is acquired (i.e. “CM1”). Then, indoor or outdoor attractions are suggested based on current weather. Finally, the user can view these attractions and select some of them for visit.

**Figure 2:** The scenarios for finding attractions and locating a restaurant
**Functional Scenario 4 (Find a Restaurant):** To find a suitable restaurant, the scenario “FS4” is used. In this scenario, the user’s food preferences are acquired. Then, a set of restaurants are suggested based on the food preferences. Finally, the user can view the restaurants’ information and select a suitable one as shown in Figure 2.B.

**Functional Scenario 5 (Notify Vehicle Speed):** During a journey, the vehicle speed may exceed the speed limit. To alert the user when this situation happens, the scenario “FS5” is used. In this scenario, the vehicle speed and the speed limit are acquired (see Figure 3). Then, based on the vehicle speed the user is either notified by the vehicle speed or alerted when the vehicle speed exceeds the speed limit.

**Figure 3:** Alerting the user when his vehicle speed exceeds the speed limit

**Figure 4:** Updating the routes to consider changes in the traffic information

**Functional Scenario 6 (Updating Routes):** The traffic information may change during a journey. To consider the traffic information changes, we introduced the scenario “FS6” shown
249

in Figure 4. This scenario is initiated in response to changes in the traffic information. Then, the selected attractions and the driving preferences are acquired to suggest suitable routes. Finally the user may select a new route for the journey to avoid congested routes for example.

1.2 The Travel Guide System’s Adaptation Scenarios

The travel guide system has 6 adaptation scenarios that specify adaptations to the 6 functional scenarios described above.

Adaptation Scenario 1 (Adapting “FS1”): The travel guide system has a number of services, and the user may include or exclude some of them. To enable the addition and removal of these services, we introduced the adaptation scenario “AS1” shown in Figure 5.A. In this scenario, the functional scenario “FS1” is adapted to only include the services that are needed (selected).

Figure 5: Adapting the functional scenarios “FS1” and “FS3”

Adaptation Scenario 3 (Adapting “FS3”): While the travel guide system in operation, the traffic information may become unavailable. To cope with such change, we have specified an adaptation fragment. This fragment removes the elements that depend on the traffic information availability from the scenario “FS3” (i.e. “AD1” in Figure 5.B). These elements are added when the traffic information is available (i.e. “AD2”). In addition, the user may have not specified his driving preferences. As such, we introduced the fragment “AD3” that adapts “FS3” to consider the changes in the driving preferences availability (see Figure 5.B).
**Adaptation Scenario 2 (Adapting “FS2”):** The functional scenario “FS2” suggests a number of attractions to the user based on the weather conditions. But, the weather information provider may be unavailable. To make the system able to work properly without the weather information, we introduced the scenario “AS2” shown in Figure 6.A. It adapts the scenario “FS2”, so that it can consider the changes in the weather information availability.

![Figure 6](image-url)  
*Figure 6:* The travel guide system’s adaptation in response to the availability of some context information

**Adaptation Scenario 4 (Adapting “FS4”):** To find a suitable restaurant, the user’s food preferences need to be taken into account as shown in the scenario “FS4”. But, the user may not define his food preferences. To cope with such situation, we introduced the scenario “AS4” (see Figure 6.B) that adapts the scenario “FS4” based on the availability of the food preferences.

**Adaptation Scenario 5 and 6 (Adapting “FS5” and “FS6”):** Similar to the above scenarios, the scenarios “AS5” and “AS6” adapt the scenarios “FS5” and “FS6” in response to changes in the availability of the speed limit and the driving preferences respectively (see Figure 7).

![Figure 7](image-url)  
*Figure 7:* Adaptations of the functional scenarios “FS5” and “FS6”
1.3 The Travel Guide System’s Properties

While the travel guide system in operation, a number of system properties need to be preserved. These properties include 13 local properties and 18 global properties.

**Functional Scenario 1:** While the functional scenario “FS1” is executing, a global property and 6 local properties need to hold. The global property specifies that the login operation should precede the logout operation (i.e. “PO1” shown in Figure 8.A). The 6 local properties specify the existence and absence of the behaviour sequences that allow the user interactions with the attractions finder, the route planner, and the restaurant locator services. Two local properties are shown in Figure 8.A. They specify the existence and absence of the “Find Attractions” sequence based on the user needs. The other four local properties are similar to these two.

![Figure 8: System properties that need to hold with the functional scenarios “FS1” and “FS3”](image)

**Functional Scenario 3:** To ensure that the “plan a route” scenario is executing properly, 2 global properties and 4 local properties need to hold. The two global properties specify that the “PlanRoute” interaction should be executed before “ProvideRoutes1” and “ProvideRoutes2” interactions (i.e. “PO1” and “PO2” shown in Figure 8.B). The four local properties specify the existence and absence of the contextual messages “CM1” and “CM3” of the functional scenario “FS3”. An example absence property is “PO3” shown in Figure 8.B. It specifies that the route planner service should not be able to acquire the traffic information (i.e. “FS3.CM1”) when this information is not available.
**Functional Scenario 2:** In response to a user request to find attractions, the travel guide system should provide indoor or outdoor attractions. Thus, two global properties are defined to specify that this sequence must hold while the system in operation. These properties are “PO1” (i.e. outdoor attractions should be provided in response to the user request to find attractions) and “PO2” (i.e. the operation “ProvideIndoorAttractions” should be executed after the request of the operation “Find Attractions”) shown in Figure 9.A. In addition, two local properties are specified to ensure the existence and absence of the contextual message “FS2.CM1” based on the availability of the weather information. An example absence property is “AB1” shown at the end of Figure 9.A.

![Figure 9: System properties that need to hold when the system suggests attractions and restaurants](image)

**Functional Scenario 4:** While the scenario “FS4” is executing, two global properties need to hold. The first property specifies that the restaurants should be provided before the user can view their information (i.e. “PO1” in Figure 9.B). The second property specifies that in response to the user request to find a restaurant, a number of restaurants are provided to him (i.e. “PO2” shown in Figure 9.B). In addition, two local properties need to hold to specify the absence and the existence of the contextual message “FS4.CM1” (see Figure 9.B).

**Functional Scenario 5:** The functional scenario “FS5” starts with acquiring the speed limit. Then, the user is updated by the vehicle speed when it is not exceeding the speed limit, while he is alerted if the vehicle speed exceeds the speed limit. To ensure that this scenario is executing properly, we specified 4 global and 2 local properties. The four global properties specify that
after the vehicle speed and the speed limit are acquired, the user is either notified by the vehicle speed or alerted when the vehicle speed goes over the speed limit. Two of these properties are shown in Figure 10.A (i.e. “PO1” and “PO2”). The two local properties specify the absence and the existence of the message “FS5.CM1” (e.g. “PO3” in Figure 10.A).

![Diagram of System Properties]

**Figure 10**: System properties that need to hold while the scenarios “FS5” and “FS6” are executing

Functional Scenario 6: While the scenario “FS6” is executing, two global properties need to hold. These two properties specify that either the interaction “ProvideRoutes1” or the operation “ProvideRoutes2” is executed in response to changes in the traffic information (i.e. “PO1” and “PO2” shown in Figure 10.B). In addition, two local properties need to be preserved. These two properties specify the absence and existence of the message “FS6.CM1” (one of these properties is shown at the end of Figure 10.B).

2. The Deployable ROAD Model of the Travel Guide System

In Chapter 5, we introduced a process to transform a system’s model designed in our notation to an executable ROAD model to enable the system realization. We also presented parts of the generated ROAD model of the travel guide system during the discussion of the transformation process. In the following, we present the complete description of the executable model that can be deployed to the ROAD runtime environment, and all files that are generated from the travel guide system model designed in our notation are available at:

<?xml version="1.0" encoding="UTF-8"?>
<tns:SMC xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tns6="http://www.swin.edu.au/ict/road/player"
xmlns:tns5="http://www.swin.edu.au/ict/road/monitor"
xsi:schemaLocation="smc smc.xsd"
xmlns:tns2="http://www.swin.edu.au/ict/road/role"
xmlns:tns1="http://www.ict.swin.edu.au/serendip/types"
name="TravelGuideSystem" dataDir="data/TravelGuideSystem"
routingRuleFile="TravelGuideSystem_routing.drl"
compositeRuleFile="TravelGuideSystem.drl">

<!--Facts-->

<Facts>
    <Fact name="UserInfo" source="External">
        <Attributes>
            <Attribute>AttractionsSelected</Attribute>
            <Attribute>SelectedFeatures</Attribute>
        </Attributes>
    </Fact>

    <Fact name="WeatherInfo" source="External">
        <Attributes>
            <Attribute>WindSpeed</Attribute>
            <Attribute>Temperature</Attribute>
            <Attribute>RainLevel</Attribute>
        </Attributes>
    </Fact>

    <Fact name="TrafficInfo" source="External">
        <Attributes>
            <Attribute>BlockRoads</Attribute>
            <Attribute>CongestedRoads</Attribute>
        </Attributes>
    </Fact>

    <Fact name="ContextInfoAvailability" source="External">
        <Attributes>
            <Attribute>TrafficInfoAval</Attribute>
            <Attribute>WeatherInfoAval</Attribute>
        </Attributes>
    </Fact>
</Facts>

<!--Contracts-->

<Contracts>
    <Contract id="FC1" type="permissive" ruleFile="TravelGuideSystem_FC1.drl">
        <Abstract>false</Abstract>
        <State>Incipient</State>
        <Terms>
            <Term id="FC1-I1" name="Login">
                <Operation name="Login">
                    <Parameters>
                        <Parameter><Type>String</Type>
                        <Name>UserName</Name></Parameter>
                        <Parameter><Type>String</Type>
                        <Name>Password</Name></Parameter>
                    </Parameters>
                    <Return>void</Return>
                </Operation>
                <Direction>AtoB</Direction>
                <Description>Description</Description>
            </Term>
            <Term id="FC1-I2" name="Logout">
                <Operation name="Logout">
                    <Parameters>
                        <Parameter><Type>String</Type>
                        <Name>User</Name></Parameter>
                    </Parameters>
                </Operation>
            </Term>
        </Terms>
    </Contract>
</Contracts>

<Contract id="FC2" type="permissive" ruleFile="TravelGuideSystem_FC2.drl">
  <Abstract>false</Abstract>
  <State>Incipient</State>
  <Terms>
    <Term id="FC2-I1" name="PlanRoute">
      <Operation name="PlanRoute">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Location</Name>
          </Parameter>
          <Parameter>
            <Type>String</Type>
            <Name>Destination</Name>
          </Parameter>
        </Parameters>
        <Return>void</Return>
      </Operation>
    </Term>
    <Term id="FC2-I2" name="ProvideRoutes1">
      <Operation name="ProvideRoutes1">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Routes</Name>
          </Parameter>
        </Parameters>
        <Return>void</Return>
      </Operation>
    </Term>
    <Term id="FC2-I3" name="ProvideRoutes2">
      <Operation name="ProvideRoutes2">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Routes</Name>
          </Parameter>
        </Parameters>
        <Return>void</Return>
      </Operation>
    </Term>
    <Term id="FC2-I4" name="SelectRoute">
      <Operation name="SelectRoute">
        <Parameters>
          <Parameter>
            <Type>int</Type>
            <Name>RouteId</Name>
          </Parameter>
        </Parameters>
        <Return>void</Return>
      </Operation>
    </Term>
  </Terms>
</Contract>
<Term>
  <RoleAID>FR1</RoleAID>
  <RoleBID>FR3</RoleBID>
  <Description>Description</Description>
</Term>

<Contract id="FC3" type="permissive" ruleFile="TravelGuideSystem_FC3.drl">
  <Abstract>false</Abstract>
  <State>Incipient</State>
  <Terms>
    <Term id="FC3-I1" name="FindAttractions">
      <Operation name="FindAttractions">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Location</Name>
          </Parameter>
        </Parameters>
        <Return> void</Return>
      </Operation>
      <Direction>AtoB</Direction>
      <Description>Description</Description>
    </Term>
    <Term id="FC3-I2" name="ProvideIndoorAttractions">
      <Operation name="ProvideIndoorAttractions">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Attractions</Name>
          </Parameter>
        </Parameters>
        <Return> void</Return>
      </Operation>
      <Direction>BtoA</Direction>
      <Description>Description</Description>
    </Term>
    <Term id="FC3-I3" name="ProvideOutdoorAttractions">
      <Operation name="ProvideOutdoorAttractions">
        <Parameters>
          <Parameter>
            <Type>String</Type>
            <Name>Attractions</Name>
          </Parameter>
        </Parameters>
        <Return> void</Return>
      </Operation>
      <Direction>BtoA</Direction>
      <Description>Description</Description>
    </Term>
    <Term id="FC3-I4" name="SelectAttraction">
      <Operation name="SelectAttraction">
        <Parameters>
          <Parameter>
            <Type>int</Type>
            <Name>AttractionId</Name>
          </Parameter>
        </Parameters>
        <Return> void</Return>
      </Operation>
      <Direction>AtoB</Direction>
      <Description>Description</Description>
    </Term>
  </Terms>
</Contract>

<Contract id="CC1" type="permissive" ruleFile="TravelGuideSystem_CC1.drl">
  <LinkedFacts>
    <Fact name="TrafficInfo"/>
  </LinkedFacts>
  <Abstract>false</Abstract>
</Contract>
<Parameter>
  <Type>String</Type>
  <Name>SelectedFeatures</Name>
</Parameter>

<Operation>
  <Return>void</Return>
  <Direction>BtoA</Direction>
  <Description>Description</Description>
</Operation>

<Term id="CC4-I2" name="getContext_UserInfo">
  <Operation name="getContext_UserInfo">
    <Parameters>
      <Parameter>
        <Type>String</Type>
        <Name>SelectedFeatures</Name>
      </Parameter>
    </Parameters>
    <Return>void</Return>
  </Operation>
  <Direction>AtoB</Direction>
  <Description>Description</Description>
</Term>

<Term>
  <RoleAID>OR1</RoleAID>
  <RoleBID>CR2</RoleBID>
  <Description>Description</Description>
</Term>

<Contract id="CC5" type="permissive" ruleFile="TravelGuideSystem_CC5.drl">
  <LinkedFacts>
    <Fact name="ContextInfoAvailability"/>
  </LinkedFacts>
  <Abstract>false</Abstract>
  <State>Incipient</State>
  <Terms>
    <Term id="CC5-I1" name="NotifyContext_TrafficInfoAval">
      <Operation name="NotifyContext_TrafficInfoAval">
        <Parameters>
          <Type>String</Type>
          <Name>TrafficInfoAval</Name>
        </Parameters>
        <Return>void</Return>
      </Operation>
      <Direction>BtoA</Direction>
      <Description>Description</Description>
    </Term>
    <Term id="CC5-I2" name="getContext_TrafficInfoAval">
      <Operation name="getContext_TrafficInfoAval">
        <Parameters>
          <Type>String</Type>
          <Name>TrafficInfoAval</Name>
        </Parameters>
        <Return>void</Return>
      </Operation>
      <Direction>AtoB</Direction>
      <Description>Description</Description>
    </Term>
    <Term id="CC5-I3" name="NotifyContext_WeatherInfoAval">
      <Operation name="NotifyContext_WeatherInfoAval">
        <Parameters>
          <Type>String</Type>
          <Name>WeatherInfoAval</Name>
        </Parameters>
        <Return>void</Return>
      </Operation>
      <Direction>BtoA</Direction>
      <Description>Description</Description>
    </Term>
  </Terms>
</Contract>
<Description>Description</Description>

<Term id="CC5-I4" name="getContext_WeatherInfoAval">
  <Operation name="getContext_WeatherInfoAval">
    <Parameters>
      <Parameter>
        <Type>String</Type>
        <Name>WeatherInfoAval</Name>
      </Parameter>
    </Parameters>
    <Return>void</Return>
  </Operation>
</Term>

<Direction>AtoB</Direction>

<Description>Description</Description>

<RoleAID>OR1</RoleAID>
<RoleBID>CR4</RoleBID>

<!-- Roles -->
<Roles>
  <Role id="CR1" name="TrafficInfo">
    <LinkedFacts>
      <Fact monitor="true" provide="true" name="TrafficInfo"/>
    </LinkedFacts>
    <Tasks>
      <tns1:Task id="GetTrafficInfo_Task" isMsgDriven="false">
        <tns1:Operation name="GetTrafficInfo"/>
        <tns1:SrcMsg contractId="CC1" termId="I1"/>
      </tns1:Task>
    </Tasks>
    <Description>Description</Description>
  </Role>
  <Role id="CR2" name="UserInfo">
    <LinkedFacts>
      <Fact monitor="true" provide="true" name="UserInfo"/>
    </LinkedFacts>
    <Tasks>
      <tns1:Task id="GetAttractionsSelected_Task" isMsgDriven="false">
        <tns1:Operation name="GetAttractionsSelected"/>
        <tns1:SrcMsg contractId="CC2" termId="I1"/>
      </tns1:Task>
      <tns1:Task id="GetSelectedFeatures_Task" isMsgDriven="false">
        <tns1:Operation name="GetSelectedFeatures"/>
        <tns1:SrcMsg contractId="CC4" termId="I2"/>
      </tns1:Task>
    </Tasks>
    <Description>Description</Description>
  </Role>
  <Role id="CR3" name="WeatherInfo">
    <LinkedFacts>
      <Fact monitor="true" provide="true" name="Weather"/>
    </LinkedFacts>
    <Tasks>
      <tns1:Task id="GetWeatherInfo_Task" isMsgDriven="false">
        <tns1:Operation name="GetWeatherInfo"/>
        <tns1:SrcMsg contractId="CC3" termId="I1"/>
      </tns1:Task>
    </Tasks>
    <Description>Description</Description>
  </Role>
  <Role id="CR4" name="ContextInfoAvailability">
    <LinkedFacts>
      <Fact monitor="true" provide="true" name="ContextInfoAvailability"/>
    </LinkedFacts>
    <Tasks>
      <tns1:Task id="GetTrafficInfoAval_Task" isMsgDriven="false">
        <tns1:Operation name="GetTrafficInfoAval"/>
        <tns1:SrcMsg contractId="CC5" termId="I1"/>
      </tns1:Task>
    </Tasks>
    <Description>Description</Description>
  </Role>
</Roles>
<tns1:Operation name="GetTrafficInfoAval">
  <tns1:SrcMsg contractId="CC5" termId="I2"/>
</tns1:Task>
</Tasks>
<Description>Description</Description>
</Role>

<Role id="FR1" name="User" >
<Tasks>
  <tns1:Task id="ProvideRoutes1_Task" isMsgDriven="false">
    <tns1:Operation name="ProvideRoutes1">
      <tns1:SrcMsg contractId="FC2" termId="I2"/>
    </tns1:Task>
  </Tasks>
</Role>

<Role id="FR2" name="Provider" >
<Tasks>
  <tns1:Task id="Login_Task" isMsgDriven="false">
    <tns1:Operation name="Login">
      <tns1:SrcMsg contractId="FC1" termId="I1"/>
    </tns1:Task>
  </Tasks>
</Role>

<Role id="FR3" name="RoutePlanner" >
<LinkedFacts>
  <Fact monitor="true" provide="false" name="TrafficInfo"/>
  <Fact monitor="true" provide="false" name="UserInfo"/>
</LinkedFacts>
<Tasks>
  <tns1:Task id="PlanRoute_Task" isMsgDriven="false">
    <tns1:Operation name="PlanRoute">
      <tns1:SrcMsg contractId="FC2" termId="I2"/>
    </tns1:Task>
  </Tasks>
</Role>
<Role id="FR4" name="AttractionsLector" >
  <LinkedFacts>
    <Fact monitor="true" provide="false" name="Weather"/>
  </LinkedFacts>
  <Tasks>
    <tns1:Task id="FindAttractions_Task" isMsgDriven="false">
      <tns1:Operation name="FindAttractions">
        <tns1:SrcMsg contractId="FC3" termId="I1"/>
      </tns1:Operation>
    </tns1:Task>
    <tns1:Task id="SelectAttractions_Task" isMsgDriven="false">
      <tns1:Operation name="SelectAttractions">
        <tns1:SrcMsg contractId="FC3" termId="I4"/>
      </tns1:Operation>
    </tns1:Task>
    <tns1:Task id="NotifyWeatherInfo_Task" isMsgDriven="false">
      <tns1:Operation name="NotifyWeatherInfo">
        <tns1:SrcMsg contractId="CC3" termId="I2"/>
      </tns1:Operation>
    </tns1:Task>
  </Tasks>
  <Description>Description</Description>
</Role>

<Role id="OR1" name="Organizer Role" >
  <LinkedFacts>
    <Fact monitor="true" provide="false" name="UserInfo"/>
    <Fact monitor="true" provide="false" name="ContextInfoAvailability"/>
  </LinkedFacts>
  <Tasks>
    <tns1:Task id="NotifySelectedFeatures_Task" isMsgDriven="false">
      <tns1:Operation name="NotifySelectedFeatures">
        <tns1:SrcMsg contractId="CC4" termId="I1"/>
      </tns1:Operation>
    </tns1:Task>
    <tns1:Task id="NotifyTrafficInfoAval_Task" isMsgDriven="false">
      <tns1:Operation name="NotifyTrafficInfoAval">
        <tns1:SrcMsg contractId="CC5" termId="I1"/>
      </tns1:Operation>
    </tns1:Task>
    <tns1:Task id="NotifyWeatherInfoAval_Task" isMsgDriven="false">
      <tns1:Operation name="NotifyWeatherInfoAval">
        <tns1:SrcMsg contractId="CC5" termId="I3"/>
      </tns1:Operation>
    </tns1:Task>
  </Tasks>
  <Description>Description</Description>
</Role>

<!-- Process Definitions -->
<ProcessDefinitions>
  <tns1:ProcessDefinition id="BS1" descr="Process definition for planning a route">
    <tns1:CoS>FC2I1PlanRouteRequested</tns1:CoS>
    <tns1:CoT>FC2I4SelectRouteCompleted</tns1:CoT>
    <tns1:BehaviourUnitRefs>
      <tns1:BehaviourUnitId>BS1_BI1_PlanRoute</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS1_BI2_GetTrafficInfoAval</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS1_BI3_GetAttractionSelected</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS1_BI4_ProvideRoutes1</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS1_BI5_ProvideRoutes2</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS1_BI6_SelectRoute</tns1:BehaviourUnitId>
    </tns1:BehaviourUnitRefs>
  </tns1:ProcessDefinition>
  <tns1:ProcessDefinition id="BS2" descr="Process definition for find attractions">
    <tns1:CoS>FC3I1FindAttractionsRequested</tns1:CoS>
    <tns1:CoT>FC3I4SelectAttractionsCompleted</tns1:CoT>
    <tns1:BehaviourUnitRefs>
      <tns1:BehaviourUnitId>BS2_BI1_FindAttractions</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS2_BI2_GetWeatherInfo</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS2_BI3_ProvideIndoorAttractions</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS2_BI4_ProvideOutdoorAttractions</tns1:BehaviourUnitId>
      <tns1:BehaviourUnitId>BS2_BI5_SelectAttractions</tns1:BehaviourUnitId>
    </tns1:BehaviourUnitRefs>
  </tns1:ProcessDefinition>
</ProcessDefinitions>
<tns1:ProcessDefinition id="BS3" descr="Process definitions for the main flow">
  <tns1:CoS>FC1I1Login</tns1:CoS>
  <tns1:CoT>FC1I2LogoutCompleted</tns1:CoT>
</tns1:ProcessDefinition>

<!-- Behaviour Units -->

<BehaviourUnits>
  <tns1:BehaviourUnit id="BS1_BI1_PlanRoute">
    <tns1:TaskRefs>
      <tns1:TaskRef id="FR3.PlanRoute_Task" preEP="FC2I1PlanRouteRequested" postEP="FC2I1PlanRouteCompleted"/>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS1_BI2_GetTrafficInfo">
    <tns1:TaskRefs>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS1_BI3_GetAttractionsSelected">
    <tns1:TaskRefs>
      <tns1:TaskRef id="CR2.GetAttractionsSelected_Task" preEP="CC1I1GetTrafficInfoCompleted" postEP="CC2I1GetAttractionsSelectedCompleted"/>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS1_BI4_ProvideRoutes1">
    <tns1:TaskRefs>
      <tns1:TaskRef id="FR1.ProvideRoutes1_Task" preEP="CC2I1GetAttractionsSelectedCompleted | AttractionsSelected" postEP="FC2I2ProvideRoutes1Completed"/>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS1_BI5_SelectRoute">
    <tns1:TaskRefs>
      <tns1:TaskRef id="FR3.SelectRoute_Task" preEP="FC2I2ProvideRoutes1Completed | FC2I3ProvideRoutes2Completed" postEP="FC2I4SelectRouteCompleted"/>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS2_BI1_FindAttractions">
    <tns1:TaskRefs>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>

  <tns1:BehaviourUnit id="BS2_BI2_GetWeatherInfo">
    <tns1:TaskRefs>
    </tns1:TaskRefs>
  </tns1:BehaviourUnit>
</BehaviourUnits>
<tns1:Constraints></tns1:Constraints>
<tns1:BehaviourUnit id="BS2_BI3_ProvideIndoorAttractions">
<tns1:TaskRefs>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS2_BI4_ProvideOutdoorAttractions">
<tns1:TaskRefs>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS2_BI5_SelectAttractions">
<tns1:TaskRefs>
<tns1:TaskRef id="FR4.SelectAttractions_Task" preEP="FC3I2ProvideIndoorAttractionsCompleted | FC3I3ProvideOutdoorAttractionsCompleted" postEP="FC3I4SelectAttractionsCompleted"/>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS3_BI1_Login">
<tns1:TaskRefs>
<tns1:TaskRef id="FR2.Login_Task" preEP="FC1I1Login" postEP="FC1I1LoginCompleted"/>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS3_BI2_BS1">
<tns1:TaskRefs>
<tns1:TaskRef id="BS1" preEP="FC1I1LoginCompleted * LoginSuccess" postEP="FC2I4SelectRouteCompleted"/>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS3_BI3_BS2">
<tns1:TaskRefs>
<tns1:TaskRef id="BS2" preEP="FC1I1LoginCompleted * LoginSuccess" postEP="FC3I4SelectAttractionsCompleted"/>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
<tns1:BehaviourUnit id="BS3_BI4_Logout">
<tns1:TaskRefs>
<tns1:TaskRef id="FR2.Logout_Task" preEP="FC2I4SelectRouteCompleted | FC3I4SelectAttractionsCompleted" postEP="FC1I2LogoutCompleted"/>
</tns1:TaskRefs>
<tns1:Constraints></tns1:Constraints>
</tns1:BehaviourUnit>
</BehaviourUnits>

<!-- Roles Players Binding -->
<PlayerBindings>
<PlayerBinding id="FP1" roleid="FR1">
<Endpoint>http://localhost:8080/axis2/services/UserGUI</Endpoint>
<Description>Description</Description>
</PlayerBinding>
<PlayerBinding id="FP2" roleid="FR2">
<Endpoint>http://localhost:8080/axis2/services/SystemProvider</Endpoint>
<Description>Description</Description>
</PlayerBinding>
<PlayerBinding id="FP3" roleid="FR3">
<Endpoint>http://localhost:8080/axis2/services/RoutePlanner1</Endpoint>
<Description>Description</Description>
</PlayerBinding>
<PlayerBinding id="FP4" roleid="FR4">
<Endpoint>http://localhost:8080/axis2/services/RoutePlanner2</Endpoint>
<Description>Description</Description>
</PlayerBinding>
</PlayerBindings>
<PlayerBinding id="FP5" roleid="FR4">
    <Endpoint>http://localhost:8080/axis2/services/AttractionsFinder</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<PlayerBinding id="CP1" roleid="CR1">
    <Endpoint>http://localhost:8080/axis2/services/RoadSideUnit</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<PlayerBinding id="CP2" roleid="CR1">
    <Endpoint>http://localhost:8080/axis2/services/TrafficInfoProvider</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<PlayerBinding id="CP3" roleid="CR2">
    <Endpoint>http://localhost:8080/axis2/services/TravelGuideDB</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<PlayerBinding id="CP4" roleid="CR3">
    <Endpoint>http://localhost:8080/axis2/services/WeatherService</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<PlayerBinding id="CP5" roleid="CR4">
    <Endpoint>http://localhost:8080/axis2/services/ContextProvidersSensor</Endpoint>
    <Description>Description</Description>
</PlayerBinding>

<OrganiserBinding>http://localhost:8080/axis2/services/OrganizerPlayer</OrganiserBinding>

<Description>The ROAD model of the travel guide system</Description>
</tns:SMC>
Appendix B: The Electronic Exam System’s Scenarios and Properties

To demonstrate the applicability of our approach, we have used it to specify the requirements and properties of an electronic exam (e-exam) software system. In Chapter 8, we have presented a set of scenarios that capture part of the system’s functional and adaptation requirements, and some properties that need to hold during the system execution. Below, we discuss the system’s scenarios and properties that are described briefly in Chapter 8.

1. Specifying the Functional Requirements of the E-exam System

As discussed in Chapter 8, the functional requirements of the e-exam system are captured by 18 functional scenarios. The scenarios 1, 3, 4, 7, and 15 are described in detail in Chapter 8. Thus, we describe the other functional scenarios below.

Functional Scenario 2 (Student Login): For a student to be able to use the system, he needs to login by following a sequence of interactions. These interactions are presented in Figure 1.A. The student provides his username and password to the accounts’ manger (i.e. “FM1”). Then, the manager checks the correctness of the student’s username and password. If the provided information is correct, a login success message is displayed to him (i.e. “FM2”). A fail message is displayed to him otherwise (i.e. “FM3”), and he needs to provide his username and password again (i.e. the sequence reference “RF1”).

![Figure 1](image_url)  
**Figure 1**: Scenarios that specify the interactions that enable a student or a lecturer to login to the e-exam software system
**Functional Scenario 8 (Lecturer Login):** The sequence of interactions that allows a student to login into the system can be used for a lecturer login. However, for some security reasons, the lecturer needs to insert his identification card into a card reader. Then, he enters his username and password which are checked for correctness and their conformance with the provided card. When the username and the password are correct, and they match the inserted card, the lecturer is successfully logged into the system as shown in Figure 1.B.

**Functional Scenario 5 (Review Answers):** After a student has finished his exam, he may want to review his answers and change them if needed. The scenario shown in Figure 2.A describes the sequence of interactions between the student and the exams’ manager to do that task. The scenario starts with providing a question and the student answer for that question (i.e. “FM1”). Then, the student modifies his answer and saves it when needed (i.e. “FM2” and “FM3”).

![Figure 2](image-url): Reviewing an exam answers and viewing the results

**Functional Scenario 6 (View Results):** The “view results” scenario shown in Figure 2.B has a set of interactions that enable a student to view an exam result by providing the subject name (i.e. “FM1” and “FM2”), or to view all his grades (i.e. “FM3”).

**Functional Scenarios 9 and 16 (Create Student and Lecturer Account):** In general, to create an account for a system’s user, he needs to provide his information which is then checked to ensure its validity. When the information is valid, an account is created and its information is provided to the user. In the e-exam software system, we have two types of users: students and lecturers. Thus, we have two functional scenarios that enable the registration of a student and a lecturer to the system. These scenarios are similar where they confirm to the general scenario of creating an account as shown in Figure 3. The main differences between the two scenarios are the functional interactions that are used to request the creation of an account (i.e. the messages
with the identifier “FM1” in Figure 3.A and Figure 3.B), and to register a student or lecturer to
the system as shown in Figure 3.A and Figure 3.B.

**Figure 3:** Creating an account for a lecturer or for a student

**Figure 4:** Designing an exam by a lecturer

**Functional Scenario 10 (Design an Exam):** The sequence of interactions that needs to be
followed by a lecturer to design an exam is shown in Figure 4. First, the lecturer requests from
the exams manager to create an exam (i.e. “FM1”), and the exams manager acquires the lecturer preferences to suggest a set of questions that match his preferences (e.g. providing multiple choice questions first). Second, the lecturer views the questions and selects suitable ones to be added to the exam (see the loop “LP1”). Finally, he previews the exam and approves it as shown at the end of Figure 4.

*Functional Scenarios 11 and 12 (Modify Questions):* To create an exam, a lecturer selects some of the pre-designed questions (as discussed above). To design such questions, the scenario shown in Figure 5.A is used. In this scenario, the lecturer is able to add a new question using the sequence reference “RF1”. This question can be one of the six types shown in Figure 5.B. The lecturer is also able to view a question, and remove or modify the question when needed using the functional messages “FM1” and “FM2” (see Figure 5.A).

![Figure 5: Specify questions to be used in designing an exam](image)

*Functional Scenarios 13 (Preview an Exam):* After selecting a set of questions to be in an exam, the lecturer needs to view the exam from the students’ perspective. The scenario shown in Figure 6.A shows the sequence of interactions to do that task. First, the lecturer selects the exam to be viewed using the sequence reference “RF1”. Second, the set of exam questions are displayed to the lecturer where he can approve a question to be in the exam, modify a question, or remove the question from the exam as shown in Figure 6.A (i.e. the loop fragment “LP1” and the parallel fragment “PR1” inside the loop).
**Figure 6**: Scenarios for previewing a designed exam and scoring an exam

**Functional Scenario 14 (Score Exam)**: To assess an exam taken by the students, the lecturer can follow the scenario presented in Figure 6.B. The lecturer first gets the list of exam papers that need to be evaluated (i.e. the functional messages “FM1” and “FM2”). Then, he selects an exam paper, gets the questions that need to evaluated manually (e.g. essay questions), and scores these questions (see the loop fragment in Figure 6.B). Finally, this process is repeated a number of times to evaluate all student papers using the sequence reference “RF2”.

**Functional Scenario 17 (View Results)**: A lecturer can view the results of an exam using the scenario shown in Figure 7.A. The lecturer can either view a summary of the results, or he can view a student’s result. He can also print these results when needed.

**Functional Scenario 18 (Select Exam)**: The e-exam system supports two types of exams: solo and group. The “solo” exam is an exam that is taken by a student. In the “group” exam, a number of students solve the exam together. In addition, the exam can be stored on a server and accessed remotely by the student during the exam (i.e. an online exam), or the exam is stored on the student computer (i.e. an offline exam). The scenario in Figure 7.B shows how a student can select an exam by choosing its type and mode by the alternative fragments “AT1” and “AT2”. Then, a set of available exams are provided to him, where he can choose one (i.e. the functional messages “FM5” and “FM6” shown in Figure 7.B).
2. Specifying the Adaptation Requirements for the E-exam System

The adaptation requirements of the e-exam system are captured using 7 adaptation scenarios as discussed in Chapter 8. The scenario “AS1” and its modification in response to unanticipated changes are described in Chapter 8. In this section, we describe the other adaptation scenarios.

Adaptation Scenario 2 and 5 (Adapting “FS4” and “FS12”): Several types of questions can be in an e-exam (e.g. true/false and multiple choices). The university may want to restrict types of questions to be in an exam. Thus, the functional scenarios which are responsible for viewing a question to a student (i.e. “FS4”) and adding a question (i.e. “FS12”) need to be adapted, so that only allowed types of questions are viewed or added. Part of the adaptation scenario that is responsible for adapting the functional scenario that views a question is shown in Figure 8.A. It contains two adaptation fragments to enable the viewing of matching and essay questions only when the university allows these types of questions. For example, the fragment “AD1” removes the optional fragment “OP3” and the functional message “FM3” when the matching question type is restricted, while these two elements are added otherwise. The scenario also contains two similar adaptation fragments to enable and disable the viewing of oral and drawing questions. Similar to the scenario “AS2”, the adaptation scenario “AS5” has fragments to restrict the types of questions that can be added to an exam by a lecturer.

Adaptation Scenario 7 (Adapting “FS18”): The functional scenario “FS18” enables a student to choose an exam type and its mode. The university may want to specify types and modes of exams.
exams to be taken by students. Thus, an adaptation scenario is specified (see Figure 8.B). It has two adaptation fragments that add or remove the “group exam” and “online exam” features from the system based on the university requirements. The fragment “AD1” removes the alternative fragment “AT1” and the message “FM2” from the scenario “FS18” when the university does not want the “group exam” feature, so that a student can only take a solo exam. The fragment also has the reverse of the two adaptation actions to make a student able to select between the two modes of the exam. The adaptation fragment “AD2” has the same structure of the fragment “AD1” to enable addition and removal of the “online exam” feature.

Figure 8: Two adaptation scenarios in the e-exam software system

*Adaptation Scenario “AS3” (Adapting “FS7”): The scenario “AS4” represents adaptations that need to be performed in response to changes in the location of a lecturer and the selected features of the system (see Figure 9). First, due to some security reasons, a lecturer is only able to score an exam while on the campus. Therefore, the adaptation fragment “AD1” removes the sequence reference “RF5” that enables a lecturer to score an exam when the lecturer is not on the campus, while it is added otherwise as shown in Figure 9. Second, the university may have a separate sub-system for registering lecturers, and then this feature is not wanted in the e-exam system. The adaptation fragment “AD2” enables the university to include or exclude this feature when needed. Third, viewing an exam result is an extra feature that may be added to (removed from) the system when needed. Thus, the adaptation fragments “AD3” is specified to make the lecturer able to view an exam’s results when this feature is selected by the university, while he is not able to do that otherwise as shown in Figure 9.*
Adaptation Scenario “AS4” (Adapting “FS10”): A lecturer may not have preferences for creating an exam’s questions. Thus, the system becomes not able to suggest questions to be used in designing an exam. To cope with the unavailability of the lecturer preferences, the adaptation scenario “AS4” is specified. Using this scenario, the lifeline “CL1” and the message “CM1” are removed from the functional scenario “FS10” when the lecturer preferences are not available, while the two elements are added otherwise as shown in Figure 10.
Adaptation Scenario “AS6” (Adapting “FS15”): At runtime, the student information may become unavailable. Thus, the scenario “FS15” needs to be adapted by removing elements that depend on the student information, while these elements are added when the user information becomes available again as shown in Figure 11. The elements include the lifeline “CL1” and the messages “CM1” and “FM1”.

![Diagram](image.png)

**Figure 11**: Adapting the e-exam system based on the availability of the user information

### 3. Specifying the Properties of the E-exam System

During the runtime execution of the e-exam system, a number of system properties need to be preserved: 38 local properties and 36 global properties (see Chapter 8). The properties related to the functional scenarios “FS1” (i.e. 7 local and 1 global), “FS3” (i.e. 4 global), and “FS15” (i.e. 2 local and 4 global) are presented in Chapter 8. Below, we describe the properties that need to hold during the execution of the other functional scenarios.

**Functional Scenario 2**: While the scenario “FS2” is executing, two properties need to be preserved. These properties are shown in Figure 12.A. They specify that in response to a student request to login to the e-exam system (i.e. “FS2.FM1”), a success (i.e. “FS2.FM2”) or a fail (i.e. “FS2.FM3”) message is sent to him.

**Functional Scenario 5**: While a student is reviewing his answers, he may change some of them. To ensure that the changed answers are saved, the message “modify a question’s answer” (i.e. “FS5.FM2”) needs to be followed by the message “save answer” (i.e. “FS5.FM3”). Thus, in Figure 12.B, a property is specified (i.e. “PO1”) that means the interaction “FS5.FM2” precedes the interaction “FS5.FM3” globally.
Figure 12: Properties that need to hold while the scenarios “FS2”, “FS5”, and “FS6” are executing

Functional Scenario 6: In response to a student request to view an exam result, his grade in the exam should be provided. The property “PO2” shown in Figure 12.B specifies the response relationship between the two tasks (i.e. “FS6.FM1” and “FS6.FM2”) that needs to be preserved while the functional scenario “FS6” is executing.

Functional Scenarios 4 and 12: While the functional scenarios “FS4” and “FS14” are executing, a set of local properties need to hold as shown in Figure 13.

Figure 13: Absence and existence properties that need to hold based on the allowed questions’ types
At runtime some elements of the scenarios “FS4” and “FS12” need to be added or removed based on questions’ types that are allowed by the university. To ensure that these two scenarios are executing as expected, for each element that can be added or removed we have two local properties. The first property ensures that the element exists in the scenario when it needs to be added, while the second property specifies that the element should not be in the scenario when it is not needed. For example, the system should not be able to view “matching questions” when it is not allowed. As such, we have two properties shown in Figure 13.A: “PO1” and “PO2”. The property “PO1” specifies that the message “FS4.FM3” should not exist in the scenario “FS4” when the matching questions are not allowed, while the property “PO2” means that the message “FS4.FM3” needs to be in the scenario “FS4” otherwise. In the same manner, the lecturer is able to add questions that are allowed only. Therefore, in Figure 13.B, we have a set of properties to specify the existence and the absence of the “add question” messages.

Functional Scenario 7: While the functional scenario “FS7” is executing, a set of properties need to be preserved. First, the property “PO1” shown in Figure 14.A specifies that the lecturer is only able to logout after a successful login.

Figure 14: Properties that need to be preserved during the execution of the scenarios “FS7” and “FS8”

Second, the local property “PO2” specifies that the lecturer needs to login into the system before he becomes able to score an exam. This property only holds when the lecturer is on the
campus. Third, at runtime some elements of the scenario “FS7” need to be removed in response to changes in the university requirements. As such, for each removable element, there are two local properties (i.e. absence and existence) as discussed above. The functional scenario “FS7” has three changeable elements, and then there are 6 local properties. One of these properties is shown in Figure 14.A (i.e. “PO3”). This property ensures that the lecture is not able to score an exam when he is not on the campus.

Functional Scenario 8: Similar to the functional scenario “FS2” (Student Login), there are two properties that need to be preserved during the execution of the scenario “FS8” (i.e. “PO2” and “PO3” shown in Figure 14.B). They specify that in response to a lecturer request to login, a success or a fail message is sent to him. In addition, another property is defined to ensure that a lecturer inserts his identification card (i.e. “FS8.FM1”) before he becomes able to login into the system (i.e. “FS8.FM2”) as shown in Figure 14.B.

Functional Scenarios 9 and 16: To ensure that the scenarios for creating an account for a lecturer (i.e. “FS9”) or for a student (i.e. “FS16”) are executing correctly, two properties need to be preserved. The first property ensures that to add a student or a lecturer, he needs to provide his details first. The second property specifies that in response to a lecturer or a student request for creating an account, a notification about the provided information validity is sent to him. Example properties that need to hold while the scenario “FS9” is executing are shown in Figure 15.A. In the same manner, two properties are specified for the scenario “FS16”.

![Functional Scenarios](image)

**Figure 15:** Properties to hold during the execution of the scenarios to create an account or an exam

Functional Scenario 10: Four system properties need to hold while the scenario “FS10” is executing, two of them are shown in Figure 15.B. The property “PO1” means the interaction
“create exam” should precede the interaction “approve exam”. The property “PO2” ensures that in response to a set of questions are suggested to the lecturer, these questions are viewed using the scenario “view question” as shown in Figure 15.B. At runtime, in response to changes in a lecturer’s preferences availability, the contextual message “CM1” (shown in Figure 10) needs to be removed or added. As such, two local properties need to be specified to ensure the existence of this message when needed and its absence otherwise.

**Functional Scenarios 11**: While a lecturer is modifying an exam’s questions, he may want to remove or modify a question. To do these two tasks, the question needs to be first displayed to the lecturer. Therefore, two properties are defined in Figure 16.A. They specify that a question needs to be displayed before the lecturer becomes able to remove (i.e. “PO1”) or modify (i.e. “PO2”) that question.

**Functional Scenario 13**: While a lecturer is viewing an exam using the scenario “FS13”, a set of properties need hold (see Figure 17.A). First, the property “PO1” means that the lecturer is only able to end an exam after starting that exam. Second, in response to a question provided

---

**Figure 16**: Precedence and response properties that need to hold while the e-exam system in operation

**Functional Scenario 17**: A lecturer may want to view an exam’s results using the scenario “FS17”. To ensure that this scenario is executing properly, we have defined two properties (see Figure 16.B): “PO1” and “PO2”. The property “PO1” specifies that when a lecturer requests a summary of an exam’s results, this summary is provided to him. The property “PO2” means in response to the interaction “FS17.FM4” (view result), the message “FS17.FM5” (provide result) is executed as shown in Figure 16.B.
by the exams manager (i.e. “FS12.FM2”), it is viewed using the sequence “view question” (i.e.
the property “PO2” shown in Figure 17.A). Third, the properties “PO3” and “PO4” specify that
in response to a question provided to the lecturer, he may approve the question to be in the exam
(i.e. “FS12.FM3”) or remove the question (i.e. “FS12.FM5”) as shown in Figure 17.A. Another
property is also specified to ensure that the lecturer is only able to modify a question when it is
provided to him.

Figure 17: Properties that need to be preserved with the scenarios “FS12”, “FS13”, and “FS18”

Functional Scenario 14: A number of properties need to hold while a lecturer is scoring an
exam (see Figure 17.B). First, when the lecturer requests a list of students to score their exam
papers, this list should be provided to him (i.e. the property “PO1” presented in Figure 17.B).
Second, the property “PO2” specifies that a student’s exam paper is provided to the lecturer (i.e.
“FS14.FM2”) only after he requests the student paper (i.e. “FS14.FM3”). Third, each question
provided to the lecturer should have a score (i.e. the property “PO3” shown in Figure 17.B) to
ensure that the exam paper is assessed completely.
**Functional Scenario 18:** In response to changes of the university needs, some elements of the scenario “FS18” are included or excluded. As such, we have four local properties that specify the absence and the existence of these changeable elements. In addition, a global property is specified for the scenario “FS18”. It specifies that to select an exam, a list of available exams need to be provided first (i.e. the property “PO4” shown in Figure 17.B).
Appendix C: The Effort Multipliers and Scale Factors of COCOMO II

The post-architecture model of COCOMO II is used to estimate the amount of effort in person-months (PM) and the time (TDEV) for the development and maintenance of a software system. This model has a number of input parameters including effort multipliers and scale factors as discussed in Chapter 9. We describe these parameters in detail below.

1. The Effort Multipliers

The post-architecture model has seventeen effort multipliers. Each multiplier is defined using a number of rating levels. The nominal level for an effort multiplier is “1.00” which does not have an effect into the estimated effort, while the other levels are below or over the nominal level to reduce or to increase the estimated effort respectively. The effort multipliers are divided into four groups: product, platform, personal, and project factors.

1.1 The Product Factors

The product factors are the factors that cause variations to the development effort because of the characteristics of the product (software system) under development. These factors are: required system reliability, database size, development for reuse, documentation of the development life cycle, and system complexity.

Required System Reliability: This factor measures to what extent the system should do its intended functionality over a period of time (i.e. the expected failure rate of the system). This factor ranges from slight inconvenience (i.e. “very low”) to risk to human life (i.e. “very high”) as shown in Table 1. The required reliability factor has different ratings based on which effort is estimated. There are rating levels that are used when the system development effort is calculated while other rating levels are used for calculating the maintenance effort that is needed to update or to repair the system (see Table 1).

Database Size: To capture the effect of the data requirements into the development effort, the database size factor is used. The effort related to the data requirements is laid in assembling and maintaining the data to test the system. The rating levels of this factor are measured relative to the ratio between the database size (D) and the software system size (P) as shown in Table 1.

Development for Reuse: To reduce the effort required to develop future software projects, a set of component are developed for reuse. The “development for reuse” factor points out the additional effort to make a set of components reusable. The components can be used in a single
project “across project”, multiple projects in a single organization “across program”, multiple organizations “across product line”, or multiple product lines “across product lines”. Table 1 shows the rating levels of this factor relative to the intended use of the reusable components.

Documentation of the Development Life Cycle: The software system needs to be documented during the development life cycle. This factor specifies the required level of documentation. The rating levels for this factor are from “many life cycle needs uncovered” to “very excessive for life cycle needs” as shown in Table 1.

Table 1: The product factors of COCOMO II

<table>
<thead>
<tr>
<th>Factor/Levels</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
<th>Extra High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability (Development)</td>
<td>slight inconvenience (0.82)</td>
<td>Low, easily recoverable loss (0.92)</td>
<td>Moderate, easily recoverable loss (1.00)</td>
<td>High financial loss (1.10)</td>
<td>Risk to human life (1.26)</td>
<td>N/A</td>
</tr>
<tr>
<td>Reliability (Maintenance)</td>
<td>slight inconvenience (1.23)</td>
<td>Low, easily recoverable loss (1.10)</td>
<td>Moderate, easily recoverable loss (1.00)</td>
<td>High financial loss (0.99)</td>
<td>Risk to human life (1.07)</td>
<td>N/A</td>
</tr>
<tr>
<td>Data Size</td>
<td>N/A</td>
<td>D/P &lt; 10 (0.90)</td>
<td>10 &lt; D/P &lt; 100 (1.00)</td>
<td>100 &lt; D/P &lt; 1000 (1.14)</td>
<td>D/P &gt; 1000 (1.28)</td>
<td>N/A</td>
</tr>
<tr>
<td>Reusability</td>
<td>N/A</td>
<td>None (0.95)</td>
<td>Across project (1.00)</td>
<td>Across program (1.07)</td>
<td>Across product line (1.15)</td>
<td>Across product lines (1.24)</td>
</tr>
<tr>
<td>Documentation</td>
<td>Many life cycle needs uncovered (0.81)</td>
<td>Some life cycle needs uncovered (0.91)</td>
<td>Right size to the life cycle needs (1.00)</td>
<td>excessive for life cycle needs (1.11)</td>
<td>very excessive for life cycle needs (1.23)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

System Complexity: To specify the system complexity, five areas need to be considered: control operations, computational operations, device-dependent operations, data management operations, and user interface management operations. Table 2 shows the rating levels of these areas and their descriptions. The rating levels range from “very low” to “extra high” based on the aspects that need to be considered in an operation.

1.2 The Platform Factors

The platform is the target machine that is used for executing the software system. There are three platform factors that need to be considered during the system development: execution time constraint, main storage constraint, and platform volatility.

Execution Time Constraint: To execute the software system, it needs some resources. The “execution time constraint” factor measures the amount of needed execution resources. The rating levels of this factor are expressed in terms of the percentage of execution resources used by the system as shown in Table 3.
Table 2: The different attributes of the system complexity factor

<table>
<thead>
<tr>
<th>Levels/Factor</th>
<th>Control Operations</th>
<th>Computational Operations</th>
<th>Device-Dependent Operations</th>
<th>Data Management Operations</th>
<th>User Interface Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Low (0.73)</td>
<td>- Code with a few non-nested structured programming.</td>
<td>- Evaluation of simple expressions.</td>
<td>- Simple read, write statements with simple formats.</td>
<td>- Simple arrays in main memory.</td>
<td>- Simple input forms, report generators.</td>
</tr>
<tr>
<td></td>
<td>- Simple module composition.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low (0.87)</td>
<td>- Nesting of structured programming operators.</td>
<td></td>
<td>- No knowledge needed of particular processor or I/O device.</td>
<td>- Single file sub-setting with no data structure changes.</td>
<td>- Use of simple graphic user interface builders.</td>
</tr>
<tr>
<td>Nominal (1.00)</td>
<td>- Simple nesting. - Inter-module control.</td>
<td>- Standard math and statistical routines.</td>
<td></td>
<td>- Multi-file input and single file output.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Simple callbacks or message passing.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High (1.17)</td>
<td>- Nested structured programming. - Queue and stack control.</td>
<td>- Basic numerical analysis.</td>
<td>- Operations at physical I/O level.</td>
<td>- Simple triggers activated by data stream contents.</td>
<td>- Widget set development and extension. - Simple voice I/O, multimedia.</td>
</tr>
<tr>
<td></td>
<td>- Distributed processing.</td>
<td>- Basic truncation, round-off concerns.</td>
<td></td>
<td>- Complex data restructuring.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Soft real-time control.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very High (1.34)</td>
<td>- Recursive coding. - Task synchronization. - Complex callbacks.</td>
<td>- Difficult but structured numerical analysis.</td>
<td>- Routines for interrupt diagnosis. - Communication line handling. - Performance intensive embedded systems.</td>
<td>- Distributed database coordination.</td>
<td>- Moderately complex 2D/3D, dynamic graphics, multimedia.</td>
</tr>
<tr>
<td></td>
<td>- Distributed processing.</td>
<td>- Simple parallelization.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Hard real-time control.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extra High (1.74)</td>
<td>- Resource scheduling with dynamically changing</td>
<td>- Difficult and unstructured numerical analysis.</td>
<td>- Device timing dependent coding. - Performance critical embedded systems.</td>
<td>- Highly coupled, dynamic relational and object structures. - Natural language data management.</td>
<td>- Complex multimedia, virtual reality, natural language interface.</td>
</tr>
<tr>
<td></td>
<td>- Microcode-level control.</td>
<td>- Complex parallelization.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Hard real-time control.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Main Storage Constraint: The “main storage constraint” factor represents the amount of storage that is needed by a software system. The rating levels of this factor are shown Table 3. These levels are expressed relative to the percentage of storage resources that are needed by the software system.

Platform Volatility: The platform means the hardware (e.g. computer hardware) and the software (e.g. compliers) elements that are used by a software system to perform its tasks. The changes to the platform range from “low” (i.e. 12 months for the major changes, and 1 month for the minor ones) to “very high” (i.e. 2 weeks for the major changes, and 2 days for the minor changes) as shown in Table 3.
Table 3: The platform factors of COCOMO II

<table>
<thead>
<tr>
<th>Factor/Levels</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
<th>Extra High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>N/A</td>
<td>N/A</td>
<td>&lt; 50% of the</td>
<td>75% of the</td>
<td>85% of the</td>
<td>95% of the</td>
</tr>
<tr>
<td>Constraint</td>
<td></td>
<td></td>
<td>available</td>
<td>available</td>
<td>available</td>
<td>available</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>resources (1.10)</td>
<td>resources (1.05)</td>
<td>resources (1.17)</td>
<td>resources (1.63)</td>
</tr>
<tr>
<td>Main storage</td>
<td>N/A</td>
<td>N/A</td>
<td>&lt; 50% of the</td>
<td>75% of the</td>
<td>85% of the</td>
<td>95% of the</td>
</tr>
<tr>
<td>Constraint</td>
<td></td>
<td></td>
<td>available</td>
<td>available</td>
<td>available</td>
<td>available</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>resources (1.00)</td>
<td>resources (1.05)</td>
<td>resources (1.17)</td>
<td>resources (1.46)</td>
</tr>
<tr>
<td>Platform Volatility</td>
<td>N/A</td>
<td>Major change:</td>
<td>Major change:</td>
<td>Major change:</td>
<td>Major change:</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 months:</td>
<td>6 months:</td>
<td>2 months:</td>
<td>2 weeks:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor Change:</td>
<td>Minor Change:</td>
<td>Minor Change:</td>
<td>Minor Change:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 months</td>
<td>2 weeks:</td>
<td>1 week:</td>
<td>2 days:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.87)</td>
<td>(1.00)</td>
<td>(1.15)</td>
<td>(1.30)</td>
<td></td>
</tr>
</tbody>
</table>

1.3 The Personal Factors

Personal factors have the greatest effect on the amount of effort required to develop a software system. These factors are used for rating the capability of the development team. They include analyst capability, programmer capability, personal continuity, application experience, platform experience, and language and tool experience.

*Analyst Capability:* The analyst team is responsible for specifying the system requirements and designing the system. The “analyst capability” factor is used for rating the ability of the analyst team to analyse the system requirements, design the system, and communicate with each others. The rating levels of this factor are shown in Table 4, which range from “very low” (i.e. only 15% of the team are able to perform the required tasks) to “very high” (i.e. 90% of the analyst team are capable of doing the analysis and the design tasks).

Table 4: The personal factors of COCOMO II model

<table>
<thead>
<tr>
<th>Factor/Levels</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
<th>Extra High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst Capability</td>
<td>15th percentile (1.42)</td>
<td>35th percentile (1.19)</td>
<td>55th percentile (1.00)</td>
<td>75th percentile (0.85)</td>
<td>90th percentile (0.71)</td>
<td>N/A</td>
</tr>
<tr>
<td>Programmer Capability</td>
<td>15th percentile (1.34)</td>
<td>35th percentile (1.15)</td>
<td>55th percentile (1.00)</td>
<td>75th percentile (0.88)</td>
<td>90th percentile (0.76)</td>
<td>N/A</td>
</tr>
<tr>
<td>Personal Continuity</td>
<td>48% / year (1.29)</td>
<td>24% / year (1.12)</td>
<td>12% / year (1.00)</td>
<td>6% / year (0.90)</td>
<td>3% / year (0.81)</td>
<td>N/A</td>
</tr>
<tr>
<td>Applications Experience</td>
<td>&lt; 2 months (1.22)</td>
<td>6 months (1.10)</td>
<td>12 months (1.00)</td>
<td>36 months (0.88)</td>
<td>72 months (0.81)</td>
<td>N/A</td>
</tr>
<tr>
<td>Platform Experience</td>
<td>&lt; 2 months (1.19)</td>
<td>6 months (1.09)</td>
<td>12 months (1.00)</td>
<td>36 months (0.91)</td>
<td>72 months (0.85)</td>
<td>N/A</td>
</tr>
<tr>
<td>Language Experience</td>
<td>&lt; 2 months (1.20)</td>
<td>6 months (1.09)</td>
<td>12 months (1.00)</td>
<td>36 months (0.91)</td>
<td>72 months (0.84)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Programmer Capability:* The “programmer capability” factor specifies the rating levels of the programmers as a team. These rating levels are shown in Table 4.
**Personal Continuity:** The “personal continuity” factor measures the annual turnover of the company employees. This factor has a number of rating levels as shown in Table 4. The “very low” rate of personal continuity indicates that 48% of the employees are leaving the company each year, while the “very high” rate shows that only 3% of the employees are leaving.

**Applications Experience:** The experience of the software development team in developing a software system is rated with the “application experience” factor. The rating levels of this factor are shown in Table 4. The “low” rating is for application experience that is less than 2 months, while the “very high” rating is for experience that is about 6 years.

**Platform Experience:** The development team experience with the platform influences the required effort where better experience with the platform reduces the development effort. Table 4 shows the rating levels for the “platform experience” factor.

**Language and Tool Experience:** The development effort of a software system is affected by the programmers experience with the programming language and its tool support that are used for implementing the system. The “language and tool experience” factor is used for rating such programmers’ experience (see Table 4 for the rating levels).

### 1.4 The Project Factors

A number of project factors influence the required effort to develop a system such as location of the development team, compression of project schedule, and the use of modern software tools.

**Location of the Development Team:** The team location affects the development effort. If the team is highly distributed, the required effort becomes large and it is reduced when the team is located in the same place (see Table 5 for the rating levels of this factor).

**Table 5:** The project factors of COCOMO II model

<table>
<thead>
<tr>
<th>Factor/Levels</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
<th>Extra High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team Location</td>
<td>International (1.22)</td>
<td>Multi-city and Multi-company (1.09)</td>
<td>Multi-city or Multi-company (1.00)</td>
<td>Same city (0.93)</td>
<td>Same building (0.86)</td>
<td>Fully collocated (0.80)</td>
</tr>
<tr>
<td>Project Schedule</td>
<td>75% of nominal (1.43)</td>
<td>85% of nominal (1.14)</td>
<td>100% of nominal (1.00)</td>
<td>130% of nominal (1.00)</td>
<td>160% of nominal (1.00)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Project Schedule:** Because of some constraints on the development of a software system, the project schedule is accelerated with respect to the nominal schedule of the system development. The acceleration is done through putting more efforts on the early phases while the later phases are parallelized. Table 5 shows the rating levels for the project schedule factor as the percentage of the schedule acceleration.
Software Tools: A number of tools are used by the development team to perform a number of engineering tasks such as requirements specification, designing the system, etc. To measure the influence of the tools into the estimated development effort of a software system, a number of attributes need to be considered including tool coverage, tool integration, and tool maturity. The rating levels for these attributes are shown in Tables 6, 7, and 8.

Table 6: The rating levels for the completeness of tool coverage

<table>
<thead>
<tr>
<th>Rating Levels</th>
<th>Completeness of Tool Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Low (1.17)</td>
<td>Text-based editor, Basic 3GL compiler, Basic library aids, Basic text-based debugger, and Basic linker</td>
</tr>
<tr>
<td>Low (1.09)</td>
<td>Graphical interactive editor, Simple design language, Simple programming library, and Simple metrics (or analysis tool)</td>
</tr>
<tr>
<td>Nominal (1.00)</td>
<td>Syntax checking, Design tools, Data transformation, and Standard support metrics</td>
</tr>
<tr>
<td>High (0.9)</td>
<td>Semantics checking, requirement specification and analysis, Design tools, and Code generator</td>
</tr>
<tr>
<td>Very High (0.78)</td>
<td>Global semantics checking, Requirement specification and analysis, Design tools with model verifier, Code generator, and Reverse engineering</td>
</tr>
<tr>
<td>Extra High (N/A)</td>
<td>Distributed requirement negotiation and trade-off, Code generator, Active repository, Specification-based static and dynamic analysis, and Pro-active project decision assistance</td>
</tr>
</tbody>
</table>

Table 7: The rating levels for the tool integration

<table>
<thead>
<tr>
<th>Rating Levels</th>
<th>Tool Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Low (1.17)</td>
<td>Individual file formats, Different user interface for each tool, and Fundamental incompatibilities between process assumptions and object semantics</td>
</tr>
<tr>
<td>Low (1.09)</td>
<td>Different file formats for each tool, Standard user interfaces among tools, and Difficult incompatibilities between process assumptions and object semantics</td>
</tr>
<tr>
<td>Nominal (1.00)</td>
<td>Customizable user interface, and Largely workable incompatibilities between process assumptions and object semantics</td>
</tr>
<tr>
<td>High (0.9)</td>
<td>Shared repository, Customizable user interface, and Largely workable incompatibilities between process assumptions and object semantics</td>
</tr>
<tr>
<td>Very High (0.78)</td>
<td>Highly associative repository, levels of different user interfaces, and Large consistency between process assumptions and object semantics</td>
</tr>
<tr>
<td>Extra High (N/A)</td>
<td>Distributed associative repository, Complete set of interfaces for different level of users, and Full consistency between process assumptions and object semantics</td>
</tr>
</tbody>
</table>

Table 8: The rating levels for the tool maturity

<table>
<thead>
<tr>
<th>Rating Levels</th>
<th>Tool Maturity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Low (1.17)</td>
<td>Version in pre-release, Simple documentation and help</td>
</tr>
<tr>
<td>Low (1.09)</td>
<td>Version is available from less than 6 Months, Up-dated documentation and help</td>
</tr>
<tr>
<td>Nominal (1.00)</td>
<td>Version is available between 1 and 2 years, Online user support</td>
</tr>
<tr>
<td>High (0.9)</td>
<td>Version is available between 2 and 3 years, Onsite technical user support</td>
</tr>
<tr>
<td>Very High (0.78)</td>
<td>Version is available more than 3 years, Expert onsite technical user support</td>
</tr>
</tbody>
</table>
2. The Scale Factors

The post-architecture model has five scale factors: precedentedness, development flexibility, risk resolution, team cohesion, and process maturity.

**Precedentedness**: The “precedentedness” factor represents the similarity between the system to be developed and the previous systems that have been developed by the development team. Based on this similarity, a rating level is chosen for the software system. Table 9 shows the rating levels of the “precedentedness” factor.

**Development Flexibility**: The software system has a set of requirements that should be met. During the system development, some of the requirements may need to be relaxed to reduce the development effort. The “development flexibility” factor specifies to what extent the system requirements can be modified (relaxed). The rating levels of this factor are shown in Table 9. It ranges from “very low” (i.e. rigorous) to “extra high” (i.e. general goals).

**Risk Resolution**: In developing a software system, there are a number of risk items. The “risk resolution” factor specifies the percentage of non risk items in the system. The rating levels for this factor are from “very low” (i.e. 20%) to “extra high” (i.e. 100%) as shown in Table 9.

**Team Cohesion**: A number of stakeholders are involved in the project development. Due to stakeholders’ cultures, difficulties in articulating their needs, and lack of experience in working as a team, the development effort of the project varies. To take into account these attributes in estimating the effort, the “team cohesion” factor is used. This factor scales from “very low” (i.e. very difficult interactions) to “extra high” (i.e. seamless interactions) as shown in Table 9.

**Process Maturity**: The post-architecture model has five scale factors: precedentedness, development flexibility, risk resolution, team cohesion, and process maturity.

<table>
<thead>
<tr>
<th>Factor/Leads</th>
<th>Very Low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very High</th>
<th>Extra High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precedentedness</td>
<td>Thoroughly unprecedented (6.20)</td>
<td>Largely unprecedented (4.96)</td>
<td>Somewhat unprecedented (3.72)</td>
<td>Generally familiar (2.48)</td>
<td>Largely familiar (1.24)</td>
<td>Thoroughly familiar (N/A)</td>
</tr>
<tr>
<td>Development Flexibility</td>
<td>Rigorous (5.07)</td>
<td>Occasional relaxation (4.05)</td>
<td>Some relaxation (3.04)</td>
<td>General conformity (2.03)</td>
<td>Some conformity (1.01)</td>
<td>General goals (N/A)</td>
</tr>
<tr>
<td>Risk Resolution</td>
<td>Little/20% (7.07)</td>
<td>Some/40% (5.65)</td>
<td>Often/60% (4.24)</td>
<td>Generally/75% (2.83)</td>
<td>Mostly/90% (1.41)</td>
<td>Full/100% (N/A)</td>
</tr>
<tr>
<td>Team Cohesion</td>
<td>Very difficult interactions (5.48)</td>
<td>Some difficult interactions (4.38)</td>
<td>Basically cooperative interactions (3.29)</td>
<td>Largely cooperative (2.19)</td>
<td>Highly cooperative (1.10)</td>
<td>Seamless interactions (N/A)</td>
</tr>
<tr>
<td>Process Maturity</td>
<td>Level 1 (Lower half) (7.80)</td>
<td>Level 1 (Upper half) (6.24)</td>
<td>Level 2 (4.68)</td>
<td>Level 3 (3.12)</td>
<td>Level 4 (1.56)</td>
<td>Level 5 (N/A)</td>
</tr>
</tbody>
</table>

---

287
Process Maturity: The process maturity is measured when the project to develop a software system starts. It assesses to what extent different engineering tasks of the system development are considered. According to the Software Engineering Institute’s Capability Maturity Model (CMM), there are five maturity levels. The rating for these maturity levels is shown in Table 9. These levels are from “very low” (i.e. lower half of level 1) to “extra high” (i.e. level 5).
Appendix D: Reduction in the Development Time and Cost

In Chapter 9, we have used the Monte Carlo simulation to estimate the average effort, cost, and time for developing and evolving the travel guide and the e-exam software systems. However, in the chapter, we have presented the graphs that show the estimated development and evolution efforts only. As such, in the following, we present the results of the estimated cost and time for the development and evolution of the two software systems.

1. The Travel Guide Software System

The estimated cost for developing the travel guide system in our approach with an average of “$43.93 K” is shown in Figure 1. When our approach is not used, the average cost is “$ 80.33 K” as shown in Figure 2. Therefore, the cost reduction equals to “45.3%”.

![Figure 1: The estimated cost for developing the travel guide software system by our approach](image)

![Figure 2: The estimated cost for developing the travel guide system when our approach is not used](image)
The development of the travel guide system by a team of five people takes “1.75” months in average when our approach is used, and it takes an average time of “3.21” when a traditional approach is used as shown in Figures 3 and 4. Therefore, the reduction in the development time obtained when our approach is used is “45.6%”.

![Figure 3: The estimated time for developing the travel guide software system by our approach](image1)

![Figure 4: The estimated time for developing the travel guide system when our approach is not used](image2)

![Figure 5: The estimated cost for evolving the travel guide software system by our approach](image3)

Using our approach to evolve the travel guide system in response to unanticipated changes costs “$ 4.92 K” in average as shown in Figure 5, while the cost is “$ 7.61 K” in average when
a traditional approach is used as shown in Figure 6. As such, the redaction in the cost needed to evolve the system is “34.3%”. In addition, a team of two persons who use a traditional approach to evolve the travel guide system takes “0.76” months compared to “0.49” months when our approach is used (i.e. the time reduction is “34.4 %”) as shown in Figures 7 and 8.

**Figure 6:** The estimated cost for evolving the travel guide system when our approach is not used

**Figure 7:** The estimated time for evolving the travel guide system when our approach is used

**Figure 8:** The estimated time for evolving the travel guide system when our approach is not used
2. The E-exam Software System

The estimated cost for developing the e-exam system in our approach is presented in Figure 9. It has an average of "$139.47 K", while the cost is "$214.96 K" when a traditional approach is used as shown in Figure 10. Thus, the reduction of the cost using our approach is "35.1 %".

![Figure 9: The estimated cost for developing the e-exam software system by our approach](image)

The development of the e-exam system using our approach takes "5.57" months in average, while it takes an average time of "8.6" when a traditional approach is used as shown in Figures 11 and 12. As such, the reduction in the development time of the e-exam system is "35.2 %".

![Figure 10: The estimated cost for developing the e-exam software system by a traditional approach](image)

![Figure 11: The estimated time for developing the e-exam software system by our approach](image)
In response to unanticipated changes, the e-exam system is evolved. The average cost of doing such evolution using our approach is “$15.88 K”, while the cost is “$24.99 K” in average when a traditional approach is used for performing the evolution task as shown in Figures 13 and 14. Consequently, the cost reduction obtained by using our approach is “36.5 %”.

**Figure 12:** The estimated time for developing the e-exam software system by a traditional approach

**Figure 13:** The estimated cost for evolving the e-exam software system by our approach

**Figure 14:** The estimated cost for evolving the e-exam software system by a traditional approach
In the case of two persons are evolving the e-exam system, they take around “1.59” months when our approach is used, while they take about “2.5” months when a traditional approach is used as presented in Figures 15 and 16. Therefore, the obtained reduction in time by using our approach is “36.4 %”.

**Figure 15:** The estimated time for evolving the e-exam software system by our approach

**Figure 16:** The estimated time for evolving the e-exam software system by a traditional approach


