Automatic MetaModel-Driven Service Component Model Generation

Xinyang Ding

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirements for the degree of Master of Engineering

2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

2 July 2024

. . . .
Date

. . . .
Ding Xinyang

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

2 July 2024

. . . .
Date

. . . .
Liu Yang

Authorship Attribution Statement

This thesis **does not** contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

2 July 2024

. . . .
Date

NTU NTU NTU NTU NTU NTU NTU NTU

. . . .
Ding Xinyang

Acknowledgements

As my student journey, especially the Master journey, draws to a close, I reflect on the experiences that have shaped this chapter of my life. Although the Master life has been a mix of challenges and joys, it has provided me with invaluable knowledge, skills, love, and friendships. Words cannot fully capture my gratitude, but I sincerely thank everyone who has supported me along the way.

Foremost, I extend my profound appreciation to Prof Liu Yang, my supervisor, whose constant support, direction, and motivation were crucial throughout my research journey. This thesis owes much to his expert knowledge and perceptive input.

I am also grateful to my colleagues at Cyber Security Lab, Nanyang Technological University, for their camaraderie and for fostering an environment of collaboration and mutual support. Special thanks to Dr Xie Xiaofei, Dr Li Yuekang, Dr Chen Honhxu, Dr Zhou Yuan, Mr Chen Kangjie, and other friends in the Cyber Security Lab, for their assistance and for sharing their knowledge and experience.

I am incredibly thankful for the steadfast backing of my family and friends throughout this journey. Their unwavering support provided me with the strength and motivation to persevere through the challenges of this journey.

Contents

Contents i

List of Figures iii

List of Tables iv

Summary v

1 Introduction 1

2 Related Work 7
- 2.1 Natural Language Processing of Textual Use Case 7
- 2.2 Metamodel for Use Cases 8
- 2.3 Model Transformation on Use Case Models 9

3 MetaModels for Use Cases and Service Components 13
- 3.1 MetamModel for Use Cases 13
  - 3.1.1 Textual Use Case 13
  - 3.1.2 Use Case Metamodel 16
- 3.2 MetaModel for Service Components 19
  - 3.2.1 Service Components 20
    - 3.2.1.1 Component Diagram 20
    - 3.2.1.2 Component Behavior 22
  - 3.2.2 Component Metamodel 25

4 Model Transformation 27
- 4.1 Transformation Rules 27
- 4.2 Correctness Checking of The Model Transformation 31

5 Component Model Generation from Metamodel Configurations: An Illustrative Example 37
- 5.1 Configuring the Use Case Metamodel: Generating a Use Case Model from a Textual Use Case 39
- 5.2 Configuring the Component Metamodel: Mapping Use Case Models to Graphical SCA Models 46

6 Experiments **49**
6.1 Description Of the Prototype 49
6.2 Experiments 50
6.3 Threat to Validity 53

7 Conclusion **54**

List of Figures

1.1 The framework of the proposed automatic metamodel-driven approach to generating service component model from requirements. 4

3.1 An example of textual use case. 15
3.2 The use case metamodel in terms of UML diagram. The diamond arrows denotes the composition associations, which mean that an object (e.g., UCMModel) has sub-objects (e.g., usecase, sud, and user). The open arrow denotes the association relationship among objects. 18
3.3 Illustrative example of SCA component 20
3.4 Illustrative example of SCA composite 21
3.5 SCA models for the CS and CL in MIS. 22
3.6 Syntax of dynamic behavior expression of a component. 23
3.7 The component metamodel. 26

4.1 Comparison of use case metamodel and component metamodel. 28
4.2 The ATL implemention for transformation rules. 30

5.1 Scope of the MIS (Market Information System). 38
5.2 Three textual use cases of MIS [1]. 39
5.3 An exmaple of parse trees under PCFG. 40
5.4 Illustration of Activity table. 41
5.5 The activity class in the use case model based on the activity table given in Table 5.1. 46
5.6 The source model of MIS (three use cases) during model transformation. 47
5.7 The target model of MIS (three components) generated from model transformation. 48

List of Tables

4.1 The relations between a use case set U and a component C 35

5.1 An example of the body of the Activity table for a concrete sentence . . 43

6.1 SCA statistics data produced by two team leaders. 50
6.2 SCA statistics data produced by our prototype. 51
6.3 The correct SCA statistics data produced by our prototype. 51
6.4 Accuracy Analysis of our prototype 51

Summary

Model-Driven Development (MDD) represents a paradigm shift in software engineering, offering numerous benefits that can lead to more efficient, reliable, and maintainable software systems. It shifts the focus from traditional coding to creating and evolving high-level abstract models, which are then transformed into executable code. With growing software complexity, MDD is expected to become a key approach in shaping future software engineering practices.

Usually, there are three levels of abstract models in MDD: Computation-independent Model (CIM), the Platform-independent model (PIM), and the Platform-specific model (PSM). CIM focuses on the description of textual system requirements of a software system, PIM concentrates on the design model of the system, abstracting away from specific implementation platforms and technologies, and PSM specifies the implementation details of the system for a particular platform or technology according to the PIM. MDD starts with the building of a CIM, then the CIM is transformed into a PIM; the PIM is subsequently used to create a PSM, ultimately leading to code generation. Model transformation among CIM, PIM, and PSM is a cornerstone of MDD, ensuring that the development process remains systematic, coherent, and aligned with both business objectives and technical requirements. Current research usually focuses on the conversion of PIMs to PSMs and subsequent code generation. The automatic transformation from CIMs to PIMs receives less attention in academic studies, largely due to several inherent challenges. First, system requirements are described in natural languages while design models are described in formal language models. Their intrinsic nature difference makes the transformation difficult. Second, there are various complex and heterogeneous software systems which make it different to design uniform transformation methods. On one hand, since requirements from different domains are different, it is hard to design a uniform method to generate CIMs for all requirements. On the other hand, due to the differences in different CIMS, it is hard to design uniform rules to do the transformation.

To partially address the aforementioned challenges, this thesis proposes a metamodel-driven method for converting CIMs into PIMs. The main idea is to extract the common features of CIMs and PIMs, based on which we define metamodels for CIMs and PIMs, and then the transformation rules between the two kinds of metamodels are defined. In this way, the generation of CIMs becomes the instantiation of the metamodel of CIMS and can be done by configuring the attributes in the metamodels; the generation of PIMs can be done based on the uniform transformation rules and instantiated CIMs. First, motivated by the factor that requirements are specified by textual use cases, we define use case metamodel for use cases that adhere to Cockburn's format. Second, we define the service component metamodel for the systems described in terms of service component architecture (SCA). SCA is a widely adopted design paradigm for service-oriented software systems. Third, according to the two kinds of metamodels, we develop a set of rules to transform use case models into SCA models. Rigorous proof is given to show the correctness of the rule set. Finally, we implement the proposed method and demonstrate its effectiveness with six software systems: Market Information System, Car Instrument Cluster System, Elevator System, Nighttime Bank Deposit System, Supermarket Checkout System, and ATM. The experimental results demonstrate that the models produced by our approach achieve greater accuracy compared to those manually created. It also works well from the perspective of different levels of a component model.

Our future research will concentrate on enhancing our proposed method along several key directions. First, we will add more sentence structure types to the current sentence base to write use cases to support more requirements, such as those described by model verbs and fuzzy words. Second, with the development of large language model (LLM) techniques, we can investigate how to use LLMs to configure the metamodel from user textual requirements. Third, since the use case metamodels will determine the software design, we will provide various types of use case metamodels, such as object-oriented use case metamodels, to support more types of system designs, such as object-oriented design.

Chapter 1

Introduction

Model-Driven Development (MDD) is an approach to designing and developing software systems. Abstract models are the fundamental elements throughout the development process. In MDD, developers focus on creating and manipulating models that represent different aspects of a software system, rather than writing code directly. MDD shows some advantages: MDD has been widely applied in various domains, such as robotics, cyber-physical systems, and autonomous vehicles.

According to the literature from Object Management Group (OMG), the abstract models can be produced from three abstraction levels, from a conceptual view down to the implementation details [2]: Computation-independent Model (CIM), Platform-independent model (PIM), and Platform-specific model (PSM). The CIM is a domain model and describes a system conceptually, independent of technological concepts and concerns. The primary target of a CIM is to understand the business context and requirements of the system's development. It allows stakeholders to focus on the problem domain and reach a consensus on the system's goals and rules before diving into technical details and implementation concerns. Statechart diagrams and activity diagrams are examples of CIMs. The PIM represents a software system's functional and behavioral characteristics, including its structure and functions, while abstaining from implementation-specific technical details. A PIM is typically created early in the software development life cycle and provides a technology-independent specification of the

system's requirements that can be used to guide the development process. The components of a PIM include the system's functional requirements, user interfaces, business logic, and other high-level aspects of the system's behavior. UML (Unified Modeling Language) diagrams, BPMN (Business Process Model and Notation) models, SCA (Service Component Architecture) models and other high-level system models used in software engineering are typical examples of PIMs. PSM is an implementation of the PIM, specifying the design of a system for a particular platform or technology. It provides a detailed blueprint for how a software system will be built on a particular platform, containing the information about the hardware, software, and operating system that will be used to develop the system. PSMs are typically created after the PIM has been developed, and code will be generated from PSMs for a specific platform. The components of a PSM may include platform-specific code, libraries, and configuration settings. PSMs can also include details about the user interface, database design, and other system components. For example, the Android platform-specific model is a PSM, including details about the Android software development kit (SDK), user interface components, and other platform-specific details, and specifies how a software system will be built on the Android operating system.

MDD starts with the building of a CIM, which is then transformed into a PIM using defined transformation rules. The PIM is subsequently converted into a PSM, and ultimately, platform-specific code is generated from the PSM. Most of the current MDD methods focus on the latter two, i.e., generating PSMs from PIMs and generating code from PSMs [3–5]. There is a scarcity of research on converting CIM to PIM automatically due to the thorough difference between the two kinds of models. However, the transformation from requirements to design is critical because, when done manually as it often is, errors are often introduced into the design model, potentially increasing the effort required for system testing. Our goal is to provide a method to mitigate this problem to some extent. Nevertheless, the transformation from CIMs to PIMs shows some challenges. First, system requirements are typically documented in textual use cases due to its concrete and descriptive nature, which proves highly effective for gathering software requirements. However, they have some drawbacks: (1) Writing use cases in natural language lacks formal syntax and semantics, providing a weak form for formal

reasoning about system behavior and features. (2) It is a lack of systematic methods to provide pivotal information for system design, such as communication channels and interfaces. These issues make the transformation from requirements to design difficult. Secondly, the complexity and heterogeneity of various software systems make it challenging to design a uniform set of transformation rules. On one hand, since requirements from different domains are different, it is hard to design a uniform method to generate CIMs for all requirements. On the other hand, due to the differences in different CIMS, it is hard to design uniform rules for the transformation.

This thesis proposes a metamodel-driven approach to automatically mapping requirements to design models, where the requirements are written in use cases, and the service component models are selected for design models. The main components of our approach include the definitions of use case metamodel and service component metamodel, metamodel-driven transformation rules that convert use case models to service component models, and the correctness checking of the transformation. Intuitively, metamodel serves as a template for corresponding concrete models, providing a high-level abstraction of the essential elements required to construct a concrete model; a specific concrete model can then be created by configuring these elements according to the given resources. It is important to note that the metamodel's definition is derived by extracting the common features shared among concrete models. Therefore, given the textual use cases in some pre-defined format, we can first apply Natural Language Processing (NLP) methods to configure the corresponding use case metamodel and generate the instantiation of use case models, then we generate the component model by configuring the component metamodel based on the use case models and transformation rules. The produced component model specifies a set of interconnected components, encapsulating their interfaces (signatures) and dynamic behaviors within the system. Specifically, motivated by the work of Ding *et al* [1], the textual use cases are formulated in Cockburn's standardized use case format, and the generated component model follows the Service Component Architecture (SCA) specification, as jointly defined by IBM and Microsoft. Cockburn's use case format is consistent with the current language categories [6] [7] [8], and can be utilized to model various requirements spanning multiple systems, such as Car Instrument Cluster System (CICS) [8, 9]. SCA provides a

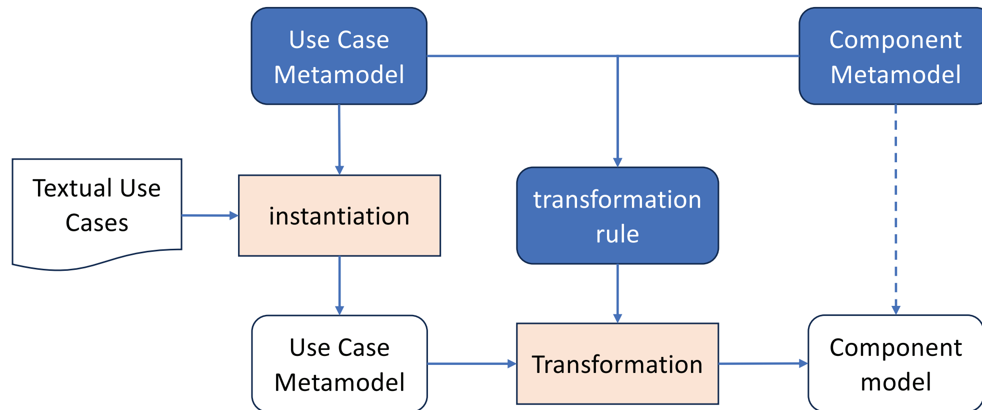


FIGURE 1.1: The framework of the proposed automatic metamodel-driven approach to generating service component model from requirements.

set of specifications for creating Service-Oriented Architecture based applications and systems. Our SCA model not only formalizes the signatures of service components but also employs a language-independent model to define the dynamic interface behavior of each service component.

Figure 1.1 shows the procedure of transforming textual use cases to an SCA model. It consists of three stages: pre-processing, model transformation, and post-processing. The first stage prepares source models for conducting model transformations. In our study, the source model is a *use case model*, an instance of the corresponding use case metamodel. The syntax analysis is conducted on sentences of textual use cases, where useful information is extracted to build *activity tables*. Next, use case models are constructed by instantiating the attributes of the corresponding metamodel using the activity tables. The transformation stage executes a program that applies a set of transformation rules to convert the model. It takes a use case model as input and generates a component model, which adheres to a component metamodel, as its output, serving as the transformed target model. At this stage, the resulting component model follows the formal definition of an SCA model. The last stage finally extracts information from the component model to derive an informal and graphical representation of an SCA model. These three stages are integrated into a prototype tool to support the automatic synthesis of an SCA model from textual use cases.

The advantages of the proposed method can summarized as follows.

- A use case metamodel can provide design information. In general, use cases do not explicitly describe design information, such as whether the system should be component-based or object-oriented. Software design usually requires some expert knowledge. In order to adopt MDD in software development, our use case metamodel provides design information such as communication interfaces among subsystems. Such design information makes our metamodel different from the existing metamodels that contain only data diagrams and control diagrams so that the existing design is manually built after model transformation.

- A use case metamodel can be configured automatically. The configured metamodel, resulting in a use case model, is an input to model transformation. Because of the difficulty of processing natural language in use cases, currently, most of the existing configurations are done manually. By using some NLP techniques, information can be extracted from the textual descriptions to automatically instantiate the use case metamodel.

- A component model can be represented in informal and formal ways. After model transformation, we obtain a component model, which can be represented by "box-arrow" like language that has no syntax and semantics [10] and by a CSP-like language that has formal syntax and semantics. With the first representation, users can understand the design easily and with the second one, a component model can be checked and analysed, and can be further used for the other model transformations to the lowest level.

The rest of this thesis is organized as follows. Chapter 2 describes the related work. Chapter 3 gives the definitions of metamodels for use cases and service components. It first analyses the features of use cases, followed by the introduction of use case metamodel. It is the source metamodel. Then, it summarizes the common attributes of a component and defines a component metamodel, which is a target metamodel. Chapter 4 defines the transformation rules to convert use case models to component models, and rigorous correctness proofing is given. Chapter 5 uses an illustrative example to show the detailed procedure to configure the metamodels and generate the service component model. Chapter 6 conducts comprehensive experiments to show the effectiveness and

efficiency of the method by comparing those manually built component models. Chapter 7 gives the conclusion of this thesis and summarizes potential avenues for future studies and development.

Chapter 2

Related Work

In this chapter, we give a brief literature review.

2.1 Natural Language Processing of Textual Use Case

Use cases are typically documented in natural language due to its concrete and descriptive nature, which proves highly effective for gathering software requirements. To create a mathematical model based on these textual use cases, it is necessary to analyze their content. In this process, Natural Language Processing (NLP) techniques are commonly employed within the field of requirements engineering [11]. With NLP techniques, there are also some well-established sentence parsers that can be used to assist the sentence analysis, such as Stanford parser [12], Apple Pie Parser [13], PC-PATR [14], XTAG Parser [15], CHILL [16], etc. In this thesis, we use Stanford parser to process the sentences. The development of this parser marked a significant milestone in 1990s natural language processing. It can interpret various plain text inputs and generate multiple analytical formats, including part-of-speech tagged text, phrase structure trees, and grammatical relations (typed dependency) format.

In order to conveniently get the execution model from use case model, the use cases must be written in a restricted language. So far, few restricted languages made it to

industrial specifications. Attempto Controlled English (ACE) is one of the most developed of such languages: "ACE appears perfectly natural, but—being a controlled subset of English—is in fact a formal language."[1] ACE is a subset of standard English with a restricted syntax and restricted semantics described by a small set of construction and interpretation rules. In this thesis, following the principle of ACE, we implemented several constraints on use case writing. The constrained language approach we employ is consistent with the classifications utilized by Breaux et al. [6], Gervasi and Zowghi [7], and Kof [8]. To aid users in crafting use cases, we also developed a text editor embedded with the StanfordParser.

2.2 Metamodel for Use Cases

To efficiently utilize MDD methodology, it is necessary to represent requirements descriptions as models. To accomplish this, we can employ a metamodel to describe requirements. However, proposing a single, universally standardized metamodel for requirements is challenging. The following are a few examples of metamodels for use cases.

In the work of Nissen *et al* [17], the authors introduced their experience of using meta-metamodel (M2-model) to address common requirements engineering challenges faced by USU's clients. The experience shows the commercial feasibility and advantages. Usually, M2-model contains four layers, arranged from top to bottom as follows: (1) The M2-model layer; (2) The modeling languages layer; (3) The layer of actual models defined using these languages; (4) The scenarios layer, which describes the application environment

OMG (Object Management Group) proposes a UML-based metamodel [18]. The metamodel's behavioral components, encompassing use cases, collaborations, and state machines, are built upon the Common Behavior package. However, these individual elements only partially cover the M2-model.

[1] http://attempto.ifi.uzh.ch/site/

Nakatani *et al* [19] proposed RD-metamodel, i.e., requirements description metamodel, based on M2-model and UML-based metamodel. It synthesizes the M2-model with metamodels for activity graphs and use cases. It introduces a flexible use case writing methodology that accommodates diverse perspectives: resource-reference, resource-structure, activity sequence, process, and actor-based approaches.

Durán et al. [20] analysed the development of the use case metamodel employed in REM, an open-source requirements management software. The initial REM metamodel [21] featured a triggering event, precondition, postcondition, and a sequence of steps outlining successful interactions, with each step containing an action and potentially a condition.

Goknil *et al.* [22] introduced a "core metamodel" for requirements models, encompassing common concepts from several widely-used methods. This metamodel can be instantiated to accommodate various requirements modeling approaches. The formalization enables reasoning about requirements, facilitating the detection of implicit relationships and inconsistencies.

Recently, Dube [23] proposed a semantic model for use case metamodel. This approach expanded the semantic scope of the UML use case metamodel, incorporating both cognitive and utility aspects. Hnatkowska and Zabawa [24] proposed the Use-Case Flow Language metamodel to improve the re-usability of the parts in use cases during requirement changes.

Unlike existing approaches, this thesis concentrates on converting CIM to PIM, offering a distinct perspective Therefore, we put some expert design knowledge into the design of the use case metamodel. Specifically, we select the service component model as the design model, our use case metamodel contains more attributes related to component models.

2.3 Model Transformation on Use Case Models

Model transformation plays a vital role in Model-Driven Development (MDD), supporting both model-to-text and model-to-model transformations [25]. The model-to-text

transformation process is essential for transforming models into textual artifacts, such as source code, reports, and documentation. Conversely, model-to-model transformations allow for the creation of different model types in various programming languages and abstraction levels based on input models. Consequently, model transformation underlies numerous activities, including refinement, synthesis, abstraction, querying, translation, migration, analysis, refactoring, normalization, optimization, merging, debugging, and synchronization [26, 27].

A model transformation is essentially a process designed to convert a model from one format or representation to another. The transformation process takes an input model, which adheres to a source form, and produces an output model that conforms to a target format. A transformation definition, written in a model transformation language, specifies the manner in which input models are transformed into output models. When the transformation language is rule-based, the definition comprises a set of transformation rules. Using this transformation definition, a transformation engine or tool generates the output model(s) from the input model(s).

Based on the representations of the input and output models of the transformation, model transformation tools can be classified into three main categories: model-to-model, model-to-text and text-to-model [25]. In this thesis, we mainly focus on model-to-model model transformation since it allows developers to work at different levels of abstraction, providing flexibility to model systems from high-level business requirements down to platform-specific designs and early verification of requirements and functionality. Therefore, we give a comprehensive literature review of the existing methods and tools.

Model-to-Model model transformation tools convert one or more input models into one or more output models. The existing approaches to model transformation can be broadly classified into two categories: relational/declarative and imperative/operational. Relational approaches define the relationships between the source and target models' objects and links [28] [29]. They concentrate on identifying the input elements to be transformed along with their related output elements, without detailing the actual execution of the transformation. An instance of a relational model transformation language is QVT Relational (QVTr), which is backed by various tools (such as Echo, QVTR-XSLT,

ModelMorf, and mediniQVT) [30]. QVT Core (QVTc) is a basic, low-level relational language that utilizes pattern matching across a set of variables [30]. In contrast, operational approaches explicitly specify the transformation process, often utilizing techniques such as graph transformation [31] [32] [33], triple graph grammars [34], and term rewriting rules [35]. This kind of methods emphasize how and when the transformation should be executed, without highlighting the necessary relationships between source and target elements. QVT Operational (QVTo) serves as an example of an imperative transformation language, similar to traditional procedural languages like C. It is supported by a range of tools, including QVTo-Eclipse, MagicDraw, SmartQVT, and Together [30].

Currently, most MDD approaches concentrate on the transformations from PIM to PSM and from PSM to code, neglecting the initial transformation from CIM to PIM [3–5]. There is a scarcity of research on automated methods for deriving system design models directly from textual requirement models. Koch *et al.* [36] presented a method for generating design models from requirements specifications through model transformation, using QVT (Query View Transformation) language to define transformation rules. Similarly, Gutiérrez *et al* [37] designed an approach that leverages model transformation, defined in the QVT-Relational language, to automatically generate an activity diagram that represents the structural use cases. Ding *et al* [1] introduced a method for transforming use case models into service component models.

However, there is still the disconnect between CIM and PIM. Use case models, while effective for describing user interactions and system requirements, do not provide the structure needed for directly defining technical models like service components. This gap necessitates a transformation process that accurately translates the semantics of natural language descriptions into the structure and constraints of formal models. Currently, there is no general method for transforming CIMs into PIMs, particularly for generating component models from use case models. Each project or domain may require a tailored approach, and without standardized transformation rules, it becomes difficult to automate the process consistently across different systems. Our work is motivated by this work but conducted from a metamodel-based view. The proposed metamodel-driven approach, with universal transformation rules, addresses the need

for adaptability, as requirements are often subject to change, and domain-specific methods may not be robust enough to accommodate diverse contexts. This adaptability allows the approach to be applied to multiple domains, supporting a range of applications and making it relevant for real-world, large-scale software systems where requirements evolve frequently.

Chapter 3

MetaModels for Use Cases and Service Components

In this chapter, we give the detailed definitions of the metamodels for use cases and service components.

3.1 MetamModel for Use Cases

Since software requirements are usually written as textual use cases [38], we focus on the situation where use case models are selected as the CIMs and investigate the transformation from the use case model. Therefore, in this section, we give the definition of metamodel for use cases.

3.1.1 Textual Use Case

Due to its comprehensive nature and ability to clearly convey the interactions between actors and the system, Cockburn's use case format [39] is selected to write use cases. It provides a structured and detailed approach to documenting use cases. First, according to Ding *et al* [1], we describe the elements included in a use case.

Definition 3.1.1. A use case usually consists of the following attributes:

- **Name** is a clear and concise name for the use case.
- **System under Discussion**, denoted as SuD, refers to the system being analysed or designed.
- **Primary actor** is the main entity (often a user or another system, rather than the system under discussion) to interact with the SuD to accomplish the goal. The use case is initiated by the primary actor.
- **Scope** defines the use case's boundaries, indicating what is included and what is excluded.
- **Main scenario**, in the form of *Main scenario: 1. step 1, 2. step 2, ..., n. step n*, defines the standard order of steps that the primary actor and the system take to fulfil the objective described by the use case. Note that the steps are numbered to show the order of operations.
- **Variation** describes alternative ways the main scenario can be accomplished without deviating significantly from the main flow.
- **Extension** captures what happens when things go wrong or when there are special conditions deviating from the main scenario.

In general, users can write a use case in any style. However, this flexibility may lead to difficulties in extracting the necessary information to construct the use case model. Therefore, in this thesis, we need to add some constraints to stipulate for the writing of use cases. Specifically, the following part provides the grammar for sentences used in the steps of use cases:

- A sentence in *Main scenario*, *Variation*, or *Extension* describes only one activity of an actor using one of the following styles, i.e.,
  - subject + verb + [object]
  - subject + verb + object + adjective
  - subject + verb + object + present participle

Use Case 3:Seller to Clerk.
Scope: MIS
SuD: CL.
Primary Actor: Seller.
preConditions:none
postConditons:none
Main Scenario:
1 Seller sends item description to the Clerk.
2 System submits information describing an item.
3 CS asks CL to input price information.
4 System asks Seller to provide price information.
5 Seller submits the price, billing andcontact information to the Clerk.
6 System enters the price, billing andcontact information to the CS.
7 CS responds with an uniquely identified acknowledgment.
8 System responds Seller with an uniquely identified authorization number.
Extensions:
2a Item not valid.
2a1 Use case aborted.

FIGURE 3.1: An example of textual use case.

- subject + verb + object + past participle
- subject + verb_1 + object_1 + to + verb_2 + [object_2]
- subject + verb + object_1 + to/from + object_2

- The names of users, subsystems, and external systems begin with capital letters. For each sentence in a use case, only the first letter of the first word is capitalized, while other words are in lower-case letters.
- The same activity should be described with the same sentence, even though in different use cases. It means that each activity is represented by a unique sentence in all use cases. Notably, while the original descriptions in different use cases may vary, the proposed sentence types allow us to transform them into a standardized format, demonstrating that this restriction is reasonable.
- Each use case has a serial number. If a use case contains the activities of other use cases, then we put the serial number at the end of the sentence in the *Main scenario*. For example, a use case has a step 'Buyer searches for an offer' that calls the Use Case #2, then this step has the representation 'Buyer searches for an offer(#2)'.

Fig. 3.1 shows a textual use case describing the interaction between a seller and the Clerk (CL) system, subsystem of the Market Information System (MIS). In this example, the seller will interact with CL, so the primary actor is Seller. The main scenario of the use case contains 8 steps. Step 2 has an extension, describing what to do if the item is not valid, i.e., the use case will be aborted.

3.1.2 Use Case Metamodel

This section will give a detailed definition of the use case metamodel. Note that the use case metamodel is used to describe the general features of use cases. It is a template consisting a set of attributes to describe use case models. These attributes in the metamodel are determined by extracting the elements in the definition of textual use cases. The instantiation of the attributes is determined by the configuration procedure of the use case metamodel based on the detailed use cases (described in Chapter 5.1). Therefore, before giving the definition of use case metamodel, we first give some basic concepts extracted from the format of textual use cases.

Definition 3.1.2 (SuD Class). SuD class aims to store the information of the system under discussion (SuD). It includes two attributes: *sudName*, describing the name of SuD, and *subs*, storing the subsystems in SuD.

Definition 3.1.3 (User Class). It is a class to describe the information of the user, which contains one attribute: *userName*.

Definition 3.1.4 (Use Case Class). A Use Case Class will be used to model use cases. Based on Definition 3.1.1, the use case class is a tuple containing seven elements:

- *uName*: the name of a use case.
- *sName*: the name of SuD described in the use case.
- *pActor*: the primary actor in the use case.
- *uActivitySet*: a string describing the activity sequence of the use case, where the activities are linked by the following operators: sequence ";", choice "or", parallel "$and-or$", and loop "b*()".
- *activities*: the activity set, where each activity in the set is an instance of the Activity Class given in Definition 3.1.5.
- *preCondition*: the conditions or states that need to be true before starting the use case. The pre-conditions must be met before initiating a use case.

- *postCondition*: the conditions or states that need to be true after completing the use case. The post-conditions must be met after successfully executing a use case.

An activity is a task or action performed by an actor in interaction with the system to accomplish the given goal. We say an activity is required (resp., provided) if a service is required (resp. provided) by the system through the activity. Interactions in the use case can be divided into synchronous and asynchronous. In the synchronous communication, the sender can proceed only when it receives a reception acknowledgement, while in the asynchronous communication, the sender can continue its execution no matter whether it receives any reception acknowledgement.

Definition 3.1.5. Activity Class: An activity can be defined by:

- *aName*: the name of an activity.
- *aType*: $aType \in \{required, provided\}$, indicating the type of an activity.
- *cType*: $cType \in \{synchronous, asynchronous\}$, indicating the type of interaction/communication.
- *sender*: the side sending a message,
- *receiver*: the side receiving a message,
- *condition*: indicating when the activity can happen,
- *isUCReference*: a label indicating whether the description represents a normal activity or references another use case.

Based on the above concepts, we can now give the definition of use case metamodel.

Definition 3.1.6. A Use Case Metamodel, denoted as UCMModel, is an abstract and uniform model to structurally describe the set of use cases in a system. It can be described by four elements:

- *sysName*: the system name for the use case metamodel.
- *SUDs*: the set of SuDs occurring in the use case.

- *users*: the set of users that can communicate with the system.

- *UCs*: the set of use cases to describe the system.

To describe the definition more intuitively, as illustrated in Figure 3.2, we give a diagram description for the use case metamodel. It means that the use case model contains an attributer of sysName, a set of use cases *UCs* described by the usecase object, a set of system under discussion *SUDs* described by the sud object, and a set of users *users* described by the user object. Each usecase object six attributers and a set of activities described by the activity class. The activity classes are derived from the steps in the Main Scenario of the textual use case.

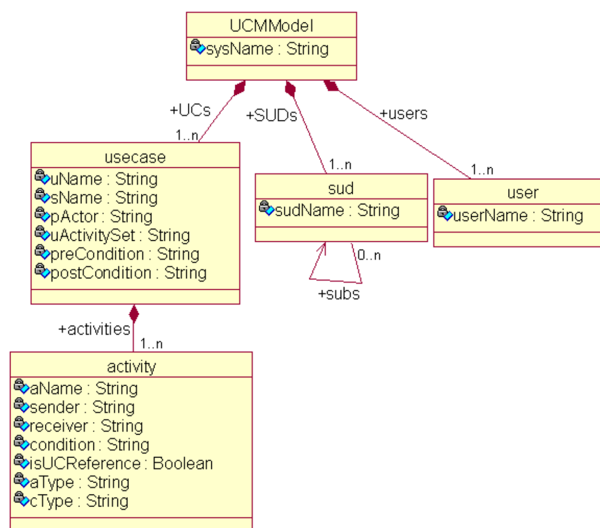


FIGURE 3.2: The use case metamodel in terms of UML diagram. The diamond arrows denotes the composition associations, which mean that an object (e.g., UCMModel) has sub-objects (e.g., usecase, sud, and user). The open arrow denotes the association relationship among objects.

For this defined use case metamodel, ensuring its completeness, consistency, and other properties generally demands advanced theorem-proving techniques and tools, which come with substantial verification costs. This is because the property verification necessitates user intervention, as the invariants directly derived from the metamodels need to be manually reinforced to build the proof. Next, we demonstrate the extent of the requirement features that this metamodel can encompass.

First, the metamodel we proposed encompasses the basic elements and their relationships. Usually, an ideal metamodel for a use case should include Activity, Subject,

Goal, Constraint, Connector, and State, as well as their relations: Dependency, Generalization, and Association. Based on the above definitions, we can easily find that our metamodel contains all of them.

Second, our metamodel covers the features of other metamodels. The following are some typical metamodels. The meta-metamodel (M2-model) from Nissen et al. [17] contains data flow diagrams and entity-relationship diagrams, which can be covered by the Activity Class in our metamodel; UML metamodel from OMG [18] contains Behavioral elements (e.g., use cases, collaborations and state machines) which can be covered by the uActivitySet in the Use Case Class of our metamodel; The metamodel from Durán et al. [20] contains triggering events, precondition, postcondition, and sequence of steps describing interactions, which can be covered by the attributes in Activity Class and uActivitySet in the Use Case Class of our metamodel.

Our metamodel shows the following novel features: (1) it contains the subsystems and their relations, specifically the description of the SuD and the relational descriptions between different SuDs; (2) it describes the relationships between different types of activities, including sequential, selection, loop, and so on; (3) it applies *cType* to identify the communication type and uses *sender* and *receiver* to indicate the direction of message flow.

3.2 MetaModel for Service Components

In this thesis, we apply the Service Component Architecture (SCA) to design the PIM since SCA provides a standardized way to define, compose, and deploy service components, ensuring consistency across different services and platforms, while other system design models may not offer a unified standard, leading to variations and inconsistencies in component definitions and interactions. In this chapter, we first give a brief introduction to SCA and then give the definition of component metamodel.

3.2.1 Service Components

SCA is a set of specifications to construct models for systems with the Service-Oriented Architecture. One of the fundamental elements in SCA is the service component, serving as a building block for service-oriented systems. These components either provide or require services, described as well-defined business functions utilizing a set of operation activities. Interactions between components are depicted through message exchanges, while business processes are illustrated through activity flows.

3.2.1.1 Component Diagram

Figure 3.3 illustrates an example of SCA components. In an SCA component, a port through which a component provides (resp., requires) a service is known as a *provided* (resp., *required*) port. Additionally, SCA ports can support two kinds of communication: *synchronous* communication, where the sender must wait for a reception acknowledgement, and *asynchronous* communication, where the sender can continue without waiting for an acknowledgement.

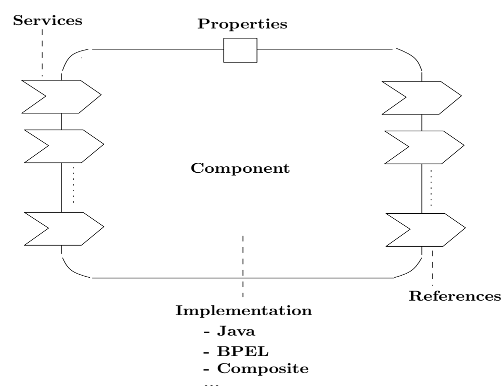


FIGURE 3.3: Illustrative example of SCA component

An SCA composite is made up of collections of service components, connections, and associated artifacts, illustrating the interconnections between these elements. Figure 3.4 shows an example of SCA composite. In a composite, a source component and a target component are connected by an SCA *wire* via a required port in the source component and a provided port in the target component. A link is referred to as a *promote*

if it extends from one provided port to another provided port or from one required port to another required port.

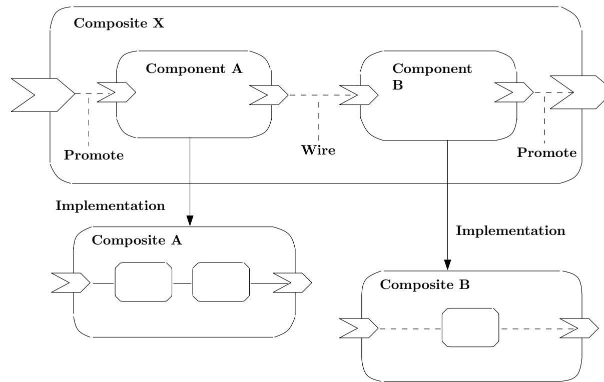


FIGURE 3.4: Illustrative example of SCA composite

According to Ding *et al* [40], we give some formal definitions related to a component. First, we give the definition of a port.

Definition 3.2.1. A port can be described as a tuple $p = (M, t, c)$, where

- M: a finite set of methods defined in p;
- t: $t = required$ or $t = provided$, indicating the port type;
- c: $c = asynchronous$ or $c = synchronous$, indicating the communication type.

Each method consists of a name op, the input parameter $\overline{T\,x}$ and the output parameter $\overline{T\,y}$, where T and x/y denote the type and names of the parameter, respectively. Therefore, a method can be formally written as $op(\overline{T\,x}\,;\,\overline{T\,y})$. Two methods are equal if they have the same input and output lists. We use $p.M$, $p.t$, and $p.c$ to denote the set of methods, port type and communication type in p, respectively. For simplicity, we use $\bullet p$ and $\blacklozenge p$ to denote a synchronous and an asynchronous port, respectively.

Definition 3.2.2. A component can be described as tuple $C = (P_p, P_r, G, W)$, where

- P_p: a finite set of provided ports;
- P_r: a finite set of required ports;
- G: a finite set of subcomponents;

- $W \subseteq (P_p \cup P_r \cup \bigcup_{C \in G} C.P_r) \times \bigcup_{C \in G}(C.P_p \cup C.P_r)$: the non-reflexive port relation, where $C.P_p$ and $C.P_r$ denote the provided and required port sets in C, respectively.

Figure 3.5 displays an example of an SCA model, which is the composition of the Computer System (CS) and Clerk (CL) in the Market Information System (MIS). Each component has a set of ports and port relations.

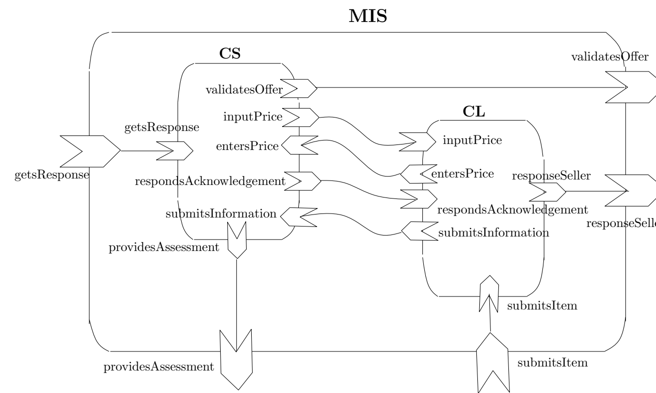


FIGURE 3.5: SCA models for the CS and CL in MIS.

3.2.1.2 Component Behavior

The component dynamic behavior are described in this section. First, the syntax of dynamic behavior expression of a component is described in Figure 3.6 [41]. The detailed meaning of each expression is described as follows.

- BBE indicates the $skip$ action and the $assign$ action $p.x = e$. The former means the component performs nothing, while the latter assigns the value of the expression e to the variable x in p.
- b represents a boolean expression.
- m represents a message;
- $p \bullet_m$ indicates that port p synchronously sends a message m, yet p is not connected to other required ports.
- $p_1 \bullet_m p_2$ means that port p_1 synchronously sends p_2 the message m.

$$
\begin{array}{lll}
BE ::= & p \bullet_m & \text{(Synchronous sending message)} \\
| & p_1 \bullet_m p_2 & \\
| & p \circ_m & \text{(Synchronous receiving message)} \\
| & p_1 \circ_m p_2 & \\
| & p \blacklozenge_m & \text{(Asynchronous sending message)} \\
| & p_1 \blacklozenge_m p_2 & \\
| & p \lozenge_m & \text{(Asynchronous receiving message)} \\
| & p_1 \lozenge_m p_2 & \\
| & \mathsf{stop} & \text{(Stop)} \\
| & BBE & \text{(Basic behavior expression)} \\
| & BE \ ; \ BE & \text{(Sequence)} \\
| & BE \lhd b \rhd BE & \text{(Condition)} \\
| & b * BE & \text{(Loop)} \\
| & BE \sqcap BE & \text{(Non-determinism)} \\
| & BE \parallel BE & \text{(Parallel)} \\
| & BE[p_1/p_2] & \text{(Renaming)} \\
| & BE[p_1 \to p_2] & \text{(Wiring)}
\end{array}
$$

FIGURE 3.6: Syntax of dynamic behavior expression of a component.

- The meanings of $\circ_m$, $\blacklozenge_m$, and $\lozenge_m$ are similar.
- $BE[p_1/p_2]$ represents syntactic renaming, wherein each occurrence of p_2 in BE is replaced by p_1. It can be employed to specify a *promote* element in an SCA composite
- $BE[p_1 \to p_2]$ defines a wiring operation; it represents a syntactic transformation, which can be described as:

$$
BE[p_1 \to p_2] =
\begin{cases}
p_1 \bullet_m p_2 & BE = p \ \bullet_m \wedge p = p_1 \\
p_2 \circ_m p_1 & BE = p \ \circ_m \wedge p = p_2 \\
p_1 \blacklozenge_m p_2 & BE = p \blacklozenge_m \wedge p = p_1 \\
p_2 \lozenge_m p_1 & BE = p \lozenge_m \wedge p = p_2 \\
BE_1[p_1 \to p_2] \ ; \ BE_2[p_1 \to p_2] & BE = BE_1 \ ; \ BE_2 \\
BE_1[p_1 \to p_2] \lhd b \rhd BE_2[p_1 \to p_2] & BE = BE_1 \lhd b \rhd BE_2 \\
b * BE_1[p_1 \to p_2] & BE = b * BE_1 \\
BE_1[p_1 \to p_2] \sqcap BE_2[p_1 \to p_2] & BE = BE_1 \sqcap BE_2 \\
BE_1[p_1 \to p_2] \parallel BE_2[p_1 \to p_2] & BE = BE_1 \parallel BE_2 \\
BE & BE = p_1 \odot p_2, where \odot \in \{\bullet_m, \circ_m, \blacklozenge_m, \lozenge_m\} \\
BBE & BE = BBE
\end{cases}
$$

Based on the above syntax, it is evident that the ports involved in message exchanges do not necessarily require identical methods. Furthermore, details regarding the communicating ports can be acquired after wiring. Note that its semantics can be described

by the classical Labelled Transition System, as given in Ding *et al* [40].

Note that wiring operations can be automatically generated in an SCA composite. It is achieved through automatic composition, which first connects the required ports and provided ports of two components and then creates a resulting composite by promoting any remaining non-wired ports.

At the signature level, two components Com_1=(P_p^1, P_r^1, G_1, W_1) and Com_2=(P_p^2, P_r^2, G_2, W_2) are *composable* if their provided porst are seperate, i.e., $P_p^1 \cap P_p^2 = \emptyset$. In the sequel, we define the automatic composition of two composable components.

First, we have the following notations. The set of automatically-related required ports can be described as:

$$\begin{aligned} \mathcal{M}_p(Com_1, Com_2) &= \{p \mid p \in P_r^1 \wedge (p.M, \textsf{provided}, p.c) \in P_p^2\} \cup \\ &\quad \{p \mid p \in P_r^2 \wedge (p.M, \textsf{provided}, p.c) \in P_p^1\}. \end{aligned}$$

Actually, $\mathcal{M}_p(Com_1, Com_2)$ is the set of ports in two components that the port in one component requires services from a port in the other component. The set of automatically generated wires can be described as:

$$\begin{aligned} \mathcal{M}_w(Com_1, Com_2) &= \{(p_1, p_2) \mid p_1 \in \mathcal{M}_p(Com_1, Com_2) \wedge \\ &\qquad p_2 = (p_1.M, \textsf{provided}, p_1.c)\}. \end{aligned}$$

Clearly, $\mathcal{M}_w(Com_1, Com_2)$ wires the ports in $\mathcal{M}_p(Com_1, Com_2)$ such that in each port pair, the first port requires the service provided by the second port. Therefore, a wire set operation is:

$$BE[W] = \begin{cases} BE, & W = \emptyset, \\ BE[p_1 \to p_2][W \setminus \{(p_1, p_2)\}], & (p_1, p_2) \in W. \end{cases}$$

Finally, we can define the automatic composition of two composable components.

Definition 3.2.3. Given two composable components $Com_1 = (P_p^1, P_r^1, G_1, W_1)$ and $Com_2 = (P_p^2, P_r^2, G_2, W_2)$, their dynamic behaviour expressions are BE_1 and BE_2, the automatic composition can be constructed as $Com \triangleq Com_1 \oplus Com_2 = (P_p, P_r, G, W)$, where

(1) $P_p = (P_p^1 \cup P_p^2) \setminus \{(p.M, \textsf{provided}, p.c) \mid p \in \mathcal{M}_p\}$;

(2) $P_r = (P_r^1 \cup P_r^2) \setminus \mathcal{M}_p$;

(3) $G = G_1 \cup G_2$;

(4) $W = \mathcal{M}_w \cup \{(p, p) \mid p \in P_p \cup P_r\}$;

(5) $BE = (BE_1 \parallel BE_2)[\mathcal{M}_w]$,

where $\mathcal{M}_p$ and $\mathcal{M}_w$ represent $\mathcal{M}_p(Com_1, Com_2)$ and $\mathcal{M}_w(Com_1, Com_2)$, respectively. Recall that the definition of port is given in Definition 3.2.1. Clearly, Com is still a component.

3.2.2 Component Metamodel

In the above section, we introduce the basic concepts of components. From these components, we can find that different systems have different components, and different components contain different ports and have different behavior. Therefore, it is difficult to automatically build a component model from the use case model for a specific system. In this section, we extract the common features of components and propose the component metamodel to model different components.

From the above definitions, the main function of a component is to provide services to others. The intrinsic characteristic of a software (sub)system is to offer services externally, so naturally, a system can naturally be considered a single, high-level component, referred to as *Component*. Additionally, since the services of a system are ultimately realized through the subsystems in the system and their interactions, thus *Component* comprises various subcomponents, defined as *innerComs*, and these subcomponents can

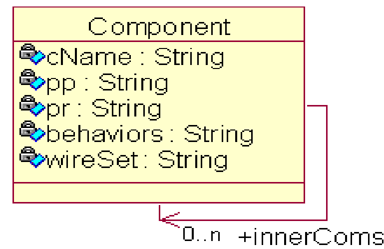


FIGURE 3.7: The component metamodel.

interact with each other. Figure 3.7 illustrates a sample Component model, represented as a UML class diagram.

According to the above analysis, we can define the metamodel of components.

Definition 3.2.4. The component metamodel, denoted as CMModel, is an abstract and uniform model to structurally describe a (sub)system in terms of components. It contains the following elements.

- cName: the name of CMModel, indicating the (sub)system's name;
- pp: the set of provided ports contained by the component,
- pr: the set of required ports contained by the component,
- behaviors: the behavior expression composed of ports and operators in SCA, including the sequence operator ;, the choice operator or, the loop operator b*(), and the parallel operator $and-or$, and so on.
- wireSet: the set of interactions between the (sub)system and its subcomponents.
- innerComs: the set of subcomponents in the component. Each subcomponent is an instance of CMModel.

Note that the functionalities of a system may depend on its internal objects and some external ones. In such a situation, interactions with external systems/objects are necessary and can be described with the provided ports (i.e., those in *pp*) and required ports (i.e., those in *pr*). Note that the generated *wireSet* will be described among the system's subcomponents as they interact with externals to realize functionalities.

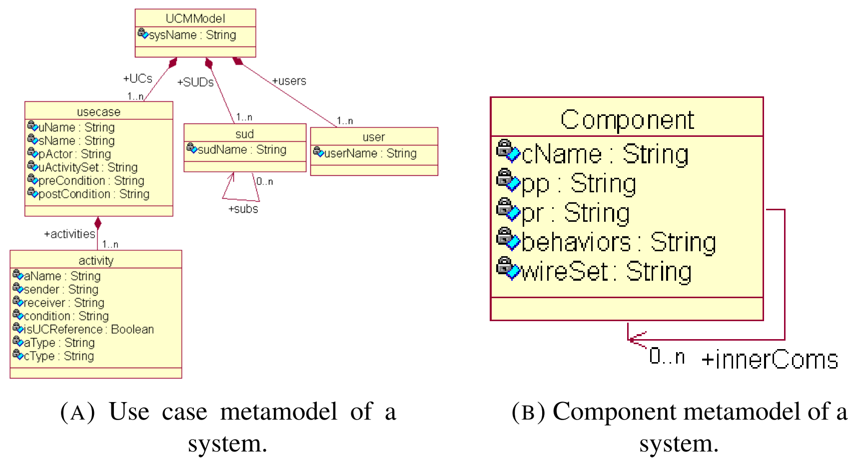
Chapter 4

Model Transformation

The previous chapter introduces the metamodels for use cases and components. The use cases models and component models are instances of the corresponding metamodels. This section gives the specific rules for transforming the use case metamodel into the component metamodel, enabling the conversion of use case models into component-based structures. Through the application of transformation rules to the use case metamodel, we can automatically generate the component model. This process instantiates the use case metamodel from the corresponding textual use case to produce a use case model, which is then transformed into a component model, leveraging the defined transformation rules. Note that the instantiation can be obtained via processing the textual use cases using well-built techniques, such as NLP methods, deep learning methods, and large language models. In this chapter, we aim to build the rule set for the transformation. The correctness of the rules is also stated in this chapter.

4.1 Transformation Rules

First of all, Figure 4.1 shows a retrospect of diagrams for the use case metamodel and component metamodel. The transformation rules are used to convert the source use case model into the target component model. To be more precise, these transformation rules configure the component metamodel based on the values from the source model. As



(A) Use case metamodel of a system.

(B) Component metamodel of a system.

FIGURE 4.1: Comparison of use case metamodel and component metamodel.

shown in Figure 4.1, component metamodel has the properties: *cName, pp, pr, behaviors*, *wireSet*, and *innerComs*. All these properties will be configured. In the following, we first give the transformation rules, and then explain their implementations in ATL (ATLAS Transformation Language)[1] in detail (as shown in Figure 4.2).

- Rule 1: Configure the component name (*cName*).

  Description: Every SuD is transformed to a component, thus the *cName* value of component comes from *sudName* of this SuD.

- Rule 2: Configure the provided/required ports of the component (*pp/pr*).

  Description: Assume SuD is mapped to a component. Either *pp* or *pr* contains a set of ports. Each port corresponds to an activity in the use case of SuD, and each port can be described by *P-aName-cType-s/r*, where *aNmae* is the activity name, *cType* is the communication type of the activity, and *s/r* is the sender or receiver.

- Rule 3: Configure the behavior expression of a component (*Behaviors*).

  Description: Assume SuD is mapped to a component. The value of *Behaviors* of the Component is the expression mapped from the activities in the *uActivitySet* of the use case with this SuD. For a single activity sequence, each activity is mapped to a port as defined in Rule 2, and the connecting operators for activities are also used as connecting operators for the ports in the target model. For multiple activity sequences, we have the following rule. Let AS, BS, CS and DS be the

[1] https://en.wikipedia.org/wiki/ATLAS_Transformation_Language

activitySets of use cases A, B, C, and D (with the same SuD), respectively, then we have

- If the postCondition of A is the preCondition of B, then the acitivytSets of A and B are connected by ';', i.e., (AS;BS).
- If A and B have the same preCondition, then their activitySets are connected by '*and-or*', i.e. *(AS and-or BS)*; If the precondition of A has no relation with that of B, their activitySets are also connected by *$and-or$*.
- If the postCondition of A is the same as the preCondition of B or C, while the preCondition of B is opposite to the preCondition of C, then their activitySets are connected by *or*, i.e. *(AS;(BS* or *CS))*.
- If A, B, C are in sequence order, the postCondition of C is the preCondition of A or D, and the preCondition of A is opposite to that of D, then a loop relation *(A-B-C-A)* is formed, and the behavior expression is *b*(AS;BS;CS);DS*.

- Rule 4: Configure the component's subcomponents (*innerComs*).

  Description: Let SuD be mapped to the component. Then all the subcomponents *innerComs* come from the sub-SuDs. Since *innerCom* is also a component, its name *cName* is the *subName* of some sub-SuD. To configure the other properties of *innerCom*, we use the same way for components.

- Rule 5: Configure the interactions among components (*wireSet*).

  Description: On one hand, the component's *wireSet* details the interactions between this component and other components, including its subcomponents and external components. On the other hand, an activity (except the activities of abort and skip) is usually related to the SuD and its subsystems or external objects. Therefore, such an activity has a wire. For simplicity, a wire is recorded as "*aName:sender→receiver*". Clearly, all these wires form *wireSet*. Since each Activity class has the attributes of sender and receiver, *wireSet* is derived from the descriptions of the activity. In the sequel, we give the process to build *wireSet*. First, retrieve the use cases of the SuD from UCMModel. Second, retrieve the Activity set of each use case; given an activity in the Activity set, if the sender

and receiver are not in this use case, a wire description is generated. Accordingly, the collection of these wire descriptions constitutes the *wireSet*.

Therefore, as shown in Figure 4.2, the rules for the transformation can be implemented with ATL.

```
116 rule sud2inner
117 {
118     from
119      sud:UCModel2!Sud
120     to
121      c:COMModel2!Component(
122         cName<-sud.sudName,
123         pp<-sud.USCModel.getTypedPortsofASUD('P',sud.sudName),
124         pr<-sud.USCModel.getTypedPortsofASUD('R',sud.sudName),
125         behaviors<-thisModule.deleteStringEnd(
126                    thisModule.getBehaviorExp(
127                    sud.USCModel.getUCSets(sud.sudName))),
128         wireSet<-thisModule.getWireSet(
129                  sud.USCModel.getUCSets(sud.sudName),
130                  sud.USCModel.allUsers),
131         innerComs<-inner
132         ),
133     inner:distinct COMModel2!Component foreach(ss in sud.subs)(
134       cName<-ss.sudName
135       )
136 }
```

FIGURE 4.2: The ATL implemention for transformation rules.

The following are the descriptions of the above code.

Line 122: Gets component name from SuD name.

Line 123: Gets the provided port set of a component, where method *getTypedPortsofASUD* generates the typed port sets for a component: For each SuD, invokes *getUCSets* to get all use cases that related to the SuD, then goes through the use case set and invokes *getPortsSet* to collect typed activities for each use case, finally invokes *getTypedPortsofASUD* to process the obtained typed port set and generate the final typed port set for the component.

Line 124: Gets the required port set of a component. The detailed process is similar to Line 123.

Line 125: Generates the behavior expression for each component, where method *deleteStringEnd* processes the behavior string generated by *getBehaviorExp*; function *getBehaviorExp* generates the behavior string for a component. SuD related use cases set is passed as a parameter, which is obtained by invoking *getUCSets*.

Line 128: Generates wire set for each component, where method *getWireSet* constructs the wire set of a component. SuD-related use cases and user set are passed as parameters.

Line 133: Gets the sub-components for each component.

Please refer to Section 5.2 for an illustration of how the proposed rules are applied to transform use case models into SCA models.

4.2 Correctness Checking of The Model Transformation

To demonstrate the correctness of the proposed model transformation, we follow Narayanan et al. [42] to verify the structural correspondences between source models (use cases models) and the resulting target models (component models). The basic intuition behind this correctness-checking strategy is that a model transformation creates certain structures of the target model by matching some structures of the source model, which can be specified by correspondences between source and target models and then be checked on the source and the resulting target models. To conduct correctness checking, we first identify the structural correspondence between use case model and the component model and then present the checking procedure.

The correctness of a model transformation can be defined based on the structural correspondences between the use case model and the component model. Let S and C denote a SuD of the UCModel and a component of a Component model, respectively. Let A denote an activity of the UCModel, and let $U = \{US_1, US_2, ..., US_n\}$ $(n \geq 1)$ be a list of use cases whose SuDs are identical. The declarations of **correctness** and

correspondences.

$$
\begin{aligned}
Correctness :&(sourceModel, targetModel) \rightarrow boolean\\
Correspondence1 :&(S, C) \rightarrow boolean\\
Correspondence2 :&(A, C) \rightarrow boolean\\
Correspondence3 :&(U, C) \rightarrow boolean
\end{aligned}
$$

For the above correspondences, each of them relates to some elements of the source and target models. Thus, we describe the relevant elements for these three correspondences below. We use **Elementsi** to denote the relevant elements for *Correspondencei* ($1 \leq i \leq 3$).

$$
\begin{aligned}
Elements1 :&S \rightarrow \{C| \quad S.sudName = C.cName\}\\
Elements2 :&A \rightarrow \{C| \quad A.aType \neq null \quad \wedge\\
&\quad (A.sender = C.cName \vee A.receiver = C.cName)\}\\
Elements3 :&U \rightarrow \{C| \quad 1 \leq i \leq n\\
&\quad \wedge \quad C.cName = US_i.sName\}
\end{aligned}
$$

In the following, we outline the procedure for checking the correctness of a model transformation.

Correctness(*sourceModel*, *targetModel*) =

($\forall$ S in *sourceModel* :

$\forall$ $C \in$ *Elements1*(S) :

Correspondence1(S,C))

and

($\forall$ $A \in$ *sourceModel* :

$\forall$ C in *Elements2*(A) :

Correspondence2(A,C))

and

($\forall$ U $\in$ *sourceModel* :

$\forall$ C in *Elements3*(U) :

Correspondence3(U,C))

Basically, given a source model and a target model, the correctness checking of the model transformation is accomplished by checking all of its three correspondences, namely, *Correspondence1*, *Correspondence2* and *Correspondence3*. The model transformation is regarded to be correct for the transformation from the given source model to the target model if all these three correspondences hold.

Next, we present the details of the three correspondences.

Correspondence1(S,C) =

(S.*sudName* = C.*cName*)

and

($\forall$ s $\in$ S.*subs* :

$\exists$ c $\in$ C.*innerComs* : c.*cName* = s.*sudName*)

Correspondence1 focuses on the correspondence relationships between suds of the source model and components of the target model. As mentioned in Section 4.1, the transformation rule 1 and rule 4 declare that a sud of the USCModel is converted to a Component of the component model, where the name and sub-SuDs of the former are mapped to the name and inner Components of the latter. In *Correspondence1*(S,C), the former correspondence is specified via (S.*sudName* = C.*cName*), and the latter one is specified as ($\forall$ s $\in$ S.*subs* : $\exists$ c $\in$ C.*innerComs* : c.*cName* = s.*sudName*).

Correspondence2(A,C) =

(A.*aType* $\neq$ *Null*) $\wedge$ ((A.*sender* = C.*cName*)

$\vee$ (A.*receiver* = C.*cName*))

and

(if (A.*aType* = ‘required’) then

C.*pr contains* ‘P-A.aName-A.aType’

else

C.*pp* contains 'P-A.aName-A.aType')

and

(C.*wireSet* contains 'A.aName: A.sender -> A.receiver')

Correspondence2 emphasizes the correspondence relationships between activities of the source model and the component information of the target model, the information of which is specified by the transformation rule 2 and rule 5. These two transformation rules declare that the port sets (*pr* and *pp*) and the wireSet of a Component are extracted from the information of the relevant activities that describe interactions between different objects. In *Correspondence2*(A,C), the properties of such an activity A is described by (A.*aType* $\neq$ *Null*) $\wedge$ ((A.*sender* = C.*cName*) $\vee$ (A.*receiver* = C.*cName*)). According to A.*aType*, the information of A is used to construct a port of C, which is a port contained in either C.*pr* or C.*pp*. That is, if (A.*aType* = 'required') then C.*pr contains* 'P-A.aName-A.aType', and C.*pp* contains 'P-A.aName-A.aType' otherwise. As a reminder, 'P-A.aName-A.aType' represents a port information of *pp* (or *pr*). Besides, the information of A is further used to construct a wire for the wireSet of C, which is described in *Correspondence2* as C.*wireSet* contains 'A.aName: A.sender -> A.receiver'.

Correspondence3 describes the relationships that should be held when creating the *behaviors* of a component from its relevant use cases, which are defined by the transformation rule 3. The detailed information of rule 3 is formalized in Table 4.1. Generally, based on different relationships among different use cases of a use case set, the resulting behavior expression of the relevant component is constructed by organizing the *uActivitySet* of these use cases in different ways. In Table 4.1, we use ***Condi*** ($1 \leq i \leq 5$) to represent different relationships among different use cases and the property of the use case set, and use ***Expi*** to denote the resulting behavior expression generated under ***Condi***. As an example for illustration, consider the second row of Table 4.1, when any two use cases satisfied ***Cond1***, the behavior expression of the relevant component will contain a sub-expression, denoted by ***Exp1***, which connects the *uActivitySet* of these two use cases by the sequence operator (namely, ';'). The definition of *Correspondence3* is given below, in which the details of ***Condi*** and ***Expi*** are declared in Table 4.1.

TABLE 4.1: The relations between a use case set U and a component C

Conditions on U	The behavior expression contained in C.*behaviors*
Cond1: U.size = 1	***Exp1***: 'US_1.*uActivitySet*'
Cond2: U.size $> 1 \wedge$ US_i.*postCondition* = US_j.*preCondition* ($1 \leq i, j \leq n$, and $i \neq j$)	***Exp2***: 'US_i.uActivitySet ; US_j.uActivitySet'
Cond3: U.size $> 1 \wedge$ US_i.*preCondition* = US_j.*preCondition* ($1 \leq i, j \leq n$, and $i \neq j$)	***Exp3***: 'US_i.uActivitySet and-or US_j.uActivitySet'
Cond4: U.size $\geq 3 \wedge$ (US_i. *postCondition* = US_j. *preCondition* $\vee$ US_i. *postCondition* = US_k. *preCondition*) $\wedge$ US_j. *preCondition* = $\neg$ US_k. *preCondition* ($1 \leq i, j, k \leq n$, and $i \neq j \neq k$)	***Exp4***: '(US_i.*uActivitySet* ; (US_j.*uActivitySet* *or* US_k.*uActivitySet*))'
Cond5: U.size ≥ 4 and US_i. *postCondition* = US_j. *preCondition* $\wedge$ US_j. *postCondition* = US_k. *preCondition* $\wedge$ (US_k. *postCondition* = US_x. *preCondition* $\vee$ US_k. *postCondition* = US_i. *preCondition*) $\wedge$ US_i. *preCondition* = $\neg$ US_x. *preCondition* ($1 \leq i, j, k, x \leq n$, and $i \neq j \neq k \neq x$)	***Exp5***: 'b*(US_i.*uActivitySet* ; US_j.*uActivitySet* ; US_k.*uActivitySet*) ; US_x.*uActivitySet*'

Correspondence3(U,C) =

(US_1.*uName* = C.*cName*)

and

(if (***Cond1***) then

C.*behaviors* contains ***Exp1***)

and

(if (***Cond2***) then

C.*behaviors* contains ***Exp2***)

and

(if (***Cond3***) then

C.*behaviors* contains ***Exp3***)

and

(if (***Cond4***) then

C.*behaviors* contains ***Exp4***)

and

(if (***Cond5***) then

C.behaviors contains ***Exp5***)

The correctness of the model transformation defined above will be verified against model transformation instances, namely, source models fed into the transformation and the relevant target models resulting from executing the transformation.

As demonstrated by the checking procedure above, there are two steps for the correctness checking of a given correspondence. The first one is to determine the relevant elements of the target model according to elements of the source model. For each of the three correspondences, an independent function is defined to accomplish this task. Specifically, the function *Elements1* focuses on the corresponce1, and this function identifies the name and inner components of a component of the target model for each given SuD of the source model. The function *Elements2* serves for the correspondence2, and it identifies port sets and the wire set of components of the target model for each given activities of the source model. Similarly, the function *Elements3* is defined for the correspondece3, and it identifies the behavior information of components of the target model for each given use case of the source model. The failing of each of these functions explicitly indicates that the given correspondence is not retained by the given source and target model and then the checking procedure ends with a negative checking result. When the identification of the relevant elements succeeds, the next step is to check the correspondence on the identified elements of the source and target model, i.e., *Correspondence1*(S, C) inspects Correspondence1 on elements S and C, where the former comes from a source model while the latter comes from the relevant target model.

After checking the above three structural correspondences on 100 executions of the proposed model transformation using 100 randomly constructed source models, it is shown that no correspondence is violated, and thus confirmed the correctness of the proposed transformation for the cases we tried.

Chapter 5

Component Model Generation from Metamodel Configurations: An Illustrative Example

In order to illustrate our method, this chapter takes the Market Information System (MIS) [43] as an illustrative example to show the usage of the proposed metamodels. It mainly contains two procedures. The first one is the generation of use case models by configuring the use case metamodels from the textual use cases, and the second one is the generation of SCA models by configuring the component metamodel using the transformation rules and the use case models.

MIS contains three subsystems and two kinds of users: The subsystems are Computer System (CS), Clerk (CL), and Supervisor (SP), and the users are Seller and Buyer [1]. Figure 5.1 shows the general architecture and business process of the system.

The requirements of MIS are described by 19 use cases: 7 use cases cover the global requirements for the system (for Seller and Buyer), 8 use cases describe the requirements for CS, and the requirements for CL and SP are described by 2 use cases, respectively. For simplicity, as illustrated in Figure 5.2, we use three of them to demonstrate our model transformation process: 1) Seller submits an offer, 2) Clerk submits an offer, and 3) Seller to Clerk.

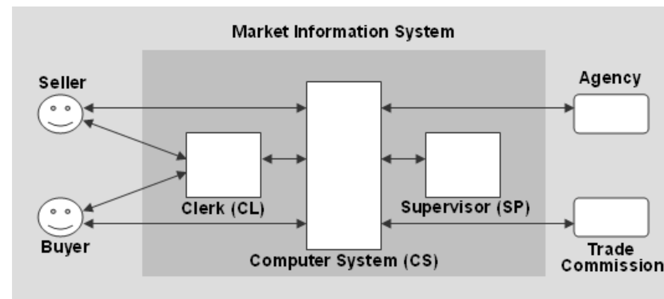


FIGURE 5.1: Scope of the MIS (Market Information System).

Remark 5.1 (Interpretation and Consistence Checking). Several methods can be used to perform interpretation and consistency checking of requirements. For examples, Uchitel *et al.* [44] demonstrate an approach that translates scenarios into behavioral specifications represented as Finite Sequential Processes. This enables the application of model checking and animation techniques to rigorously analyse the specified behavior. Damas *et al.* [45] generate and check a labeled transition system from Message Sequence Charts (MSC) [46]. The work [47] applies linear temporal logic to generate and verify goal specifications from positive and negative scenarios described in MSC-like form. Given that our restricted language style closely resembles that used in Kof [8], we can systematically convert textual use cases into MSC by applying the same rules, enabling the interpretation and consistency verification of the specified interactions. During the interpretation process, we should determine the following options: Pre-arity, Post-arity, Immediacy, Precedency, Nullity, and Repeatability. The method proposed in Alur and Yannakakis [48] can be applied to verify the consistency. This method involves creating an automaton that recognizes all possible linearizations of the partial order defined by an MSC. The generated automaton is then checked against a property automaton, enabling the verification of the specified property against the MSC's partial order behavior.

Use Case 1: Seller submits an offer.
Scope: Marketplace.
SuD: MIS.
Primary Actor: Seller.
preConditions:none
postConditons:none
Main Scenario:
1 Seller sends item description.
2 System validates the description.
3 System asks Seller to provide price information.
4 Seller submits price and enters contact and billing information.
5 System validates Seller's contactinformation.
6 System verifies the Seller's history to permit the seller to operate.
7 System validates the whole offer with the Trade Commission.
8 Trade Commission gives response tosystem.
9 System lists the offer in published offers.
10 System responds Seller with a uniquely identified authorization number.
Extensions:
2a Item not valid.
2a1 Use case aborted.
6a Seller's history inappropriate.
6a1 Use case aborted.
8a Trade commission rejects the offer.
8a1 Use case aborted.
Variations:
2b Price assessment available.
2b1 System provides a price assessment.

Use Case 2: Clerk submits an offer.
Scope: MIS
SuD: CS.
Primary Actor:CL.
preConditions:none
postConditons:none
Main Scenario:
1 Clerk submits information describing anitem.
2 System validates the description.
3 System asks CL to input price information.
4 Clerk enters price and billing information.
5 System validates Seller's contact information.
6 System asks the Supervisor to validatethe seller.
7 Supervisor permits the seller to operate.
8 System validates the whole offer with the Trade Commission.
9 Trade Commission gives response to system.
10 System lists the offer in published offers.
11 System responds an uniquely identified acknowledgment to CL.
Extensions:
2a Item not valid.
2a1 Use case aborted.
7a History is not appropriate
7a1 Use case aborted.
9a Trade commission rejects the offer.
9a1 Use case aborted.
Variations:
3b Price assessment available.
3b1 System provides a price assessment.

Use Case 3:Seller to Clerk.
Scope: MIS
SuD: CL.
Primary Actor: Seller.
preConditions:none
postConditons:none
Main Scenario:
1 Seller sends item description to the Clerk.
2 System submits information describing an item.
3 CS asks CL to input price information.
4 System asks Seller to provide price information.
5 Seller submits the price, billing andcontact information to the Clerk.
6 System enters the price, billing andcontact information to the CS.
7 CS responds with an uniquely identified acknowledgment.
8 System responds Seller with an uniquely identified authorization number.
Extensions:
2a Item not valid.
2a1 Use case aborted.

FIGURE 5.2: Three textual use cases of MIS [1].

5.1 Configuring the Use Case Metamodel: Generating a Use Case Model from a Textual Use Case

The source model is the input for the model transformation. To get the source model, we need to configure the use case metamodel defined in Chapter 3.1.2. Recall that there are

five classes represented in a use case metamodel: UCMModel, Use Case, SuD, User, and Activity, and each class has attributes. Therefore, we need to instantiate these attributes, whose values can be determined from the textual use cases. The configuration process contains two steps. The first step is to get the required values from the analysis of the textual use cases, and these values are stored in Activity tables, where each Activity table corresponds to one use case. In this step, we apply StanfordParser[1] to perform sentence analysis. In detail, StanfordParser can generate a detailed contextual structure for each sentence in the textual use case. In the sequel, we take the sentence "*1 Seller submits item description*" as an example to show the functionality of StanfordParser.

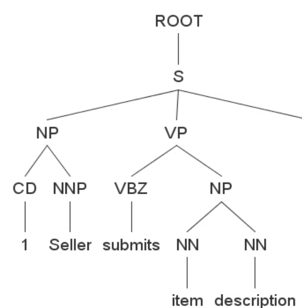


FIGURE 5.3: An exmaple of parse trees under PCFG.

Figure 5.3 displays the PCFG (probabilistic context-free grammar) structure generated from the Stanford Parser. In StanfordParser, the output of this structure is a text file, which is given as follows:

```
(ROOT
  (S
    (NP (CD 1) (NN Seller))
    (VP (VBZ submits)
      (NP (NN item) (NN description)))
    (...)))
```

According to the generated output text, we can easily retrieve the activity data of each use case.

The second step is to configure the use case metamodel with the obtained Activity data. The following gives the details.

[1]http://nlp.stanford.edu/

STEP 1: Activity Table Construction. To automatically configure the attributes, we first build an Activity table for each use case according to the data extracted from the steps in the Main scenario of the use case. As shown in Fig 5.4, the table mainly contains two parts: the header and the body. The descriptions of the features in the header are the same as the use case. The descriptions of the body part are given as follows.

Use Case	SuD	Primary Actor	Scope	preCondition	postCondition

(a) The header of an Activity table

No	Activity-Name	Activity-Type	Comm-Type	Sender	Receiver	condition	isUCReference

(b) The body of an Activity table

FIGURE 5.4: Illustration of Activity table.

- *No*: It is the serial number of the steps in main scenario, variation, and extension in the use case. For example, the *No* of the activity *1 Seller sends item description.* is 1.

- *Activity-Name*: It is the activity name. There are two situations. (1) If the sentence contains *abort* or *go to step i*, then the *activity-name* is *abort* or *step i*. They are special activities. The *Activity-Name* of the activity *Use case aborted* is *abort*. (2) Otherwise, the description of the *Activity-Name*, denoted as *activityName*, will consist of the first verb and the first noun after the verb. For example, the *Activity-Name* of the activity *1 Seller sends item description.* is *sendsitem*. If an activity references some activities in another use case, then its description will be tagged by the serial number of that use case. For example, the sentence "Buyer searches for an offer(#2)" means that this activity refers to all activities contained in use case #2. Hence, the name of the activity is denoted by "UC+serial number".

- *Sender/Receiver*: There are also two situations for the descriptions of sender and receiver. (1) When "System" is the subject, the sender is the System under Development (SuD) in the use case, and the receiver is either the direct object of the

sentence (if present) or null (indicating an internal activity). (2) When the subject is not “System”, the sender is the subject of the sentence, and the receiver is the System under Development (SuD) in the use case. For example, in the first activity of Use Case 1 in Figure 5.2 (i.e., *1 Seller sends item description.*), the sender is *Seller*, and the receiver is SuD, i.e., MIS.

- *condition*: It is pre-conditions under which an activity can happen. In general, the pre-conditions for extension and variation are explicitly declared. For example, let’s consider the following extension.

```
Extensions:
  1a The buyer did not find any matching offer
      1a1 Use case aborted
```

In this extension, the condition for the happening of the activity “1a1” is *cond=The buyer did not find any matching offer*. Since the activity “1a1” is the only exception to the activity “1”, we imply that the condition of activity “1” is *!cond*.

- *isUCReference*: If the activity name is given as ’UC+serial number’, then the attribute *isUCReference* of this activity is set to ’true’, otherwise this attribute value is ’false’.

- *Activity-Type* and *Comm-Type*: To determine *Activity-Type* and *Comm-Type*, we will first create a stack to store the body data structure, which consists of eight attributes: *Serial Number, Activity-Name, Sender, Receiver, condition, isUCReference, Activity-Type, Comm-Type*. We push into the stack the activities of a use case one by one on their orders. Let *AT* be the activity to be pushed into the stack, where *Sender* = *aS*, *Receiver* = *aR*. Then, its *Activity-Type* and *Comm-Type* can be determined as follows.

  – If the stack is empty, push *AT* onto the stack. Then, determine the *Comm-Type* and *Activity-Type* as follows: (1) If *AT* is the last activity, set *Comm-Type* = *asynchronous*; (2) If Sender = SuD, set *Activity-Type* = *required*; otherwise, set *Activity-Type* = *provided*.

  – If the stack is not empty, consider the top element, denoted as *ST*. Suppose the *Sender* and *Receiver* of *ST* are *sS* and *sR*, respectively. There are two

Algorithm 1: Activiy Table Contructure for A Use Case

Input: Textual use cases.
Output: Activity tables.

```
1  Read use cases from text file: UCs = ReadFile();
2  Process the use cases using the StanfordParser: TUCs = StanfordParser(UCs);
3  pos = 0;
4  current_UC = getOneUC(TUCs, pos), next_UC = getOneUC(TUCs, current_UC) ;
5  while current_UC do
6      Get the sentence lines in the use case: sentence_process(current_UC);
7      for each line line_i do
8          if the corresponding sentence belongs to (Main scenario || Variation || Extension) then
9              if "abort" in line_i then
10                 str = getNumber(line_i)+ "abort"
11             else if "go to step" in line_i then
12                 str = getNumber(line_i)+ "stepj"
13             else
14                 str = getNumber(line_i) + getFirstVB(line_i) + getNN(line_i) +
                     getType(line_i) + getComType(line_i) + getSender(line_i) + getReceiver(line_i);
15         else
16             str = getSentence(line_i)
17         str+ = \n ;
18         write str to the output file;
19     current_UC = getOneUC(TUCs, next_UC), next_UC = getOneUC(TUCs, current_UC) ;
```

situations. (1) *aS=sS* or (aS=sR and aR≠sS). In this situation, we have *Comm-type=asynchronous*; if *Sender=SuD*, set *Activity-Type = required*; otherwise, set *Activity-Type = provided*. Remove *ST* from the top of the stack and replace it with *AT*. (2) *aS=tR* and *aR=tS*. In this situation, *Comm-Type = synchronous*; if *tS=SuD*, then we set *Activity-Type = required*; otherwise, set *Activity-Type = provided*. Pop *ST* off the stack.

Note that similarly, *Activity-Type* and *Comm-Type* for the activities in Extension and Variation can be determined using the same steps outlined above. *Activity-Type* and *Comm-Type* for the abort activity are both *Null*.

TABLE 5.1: An example of the body of the Activity table for a concrete sentence

1	sendsItem	provided	asynchronous	Seller	MIS	NULL	false

Algorithm 1 describes the structure of the Activity Table for a use case. As an illustrative example, applying Algorithm 1 to the first activity of Use Case 1 in Figure 5.2, i.e., "*1 Seller sends item description*", we can get its activity data, as shown in Table 5.1. The value of *No* attribute is 1. The value of *Activity-Name* attribute is *sendsItem*, consisting of the first verb and the first noun. The values of *Sender* and *Receiver* are *Seller* and *MIS* (i.e., SuD) since the subject is Seller. The value of *Activity-Type* is *provided* since the $Sender$ is not the SuD. The value of *Comm-Type* is *asynchronous*

since it is the first activity, meaning that aS=sS, which satisfies the first item of the second situation in the determination of *Activity-Type* and *Comm-Type*. There are no conditions or reference use cases.

STEP 2. Building Source Model. The second step is to configure the attributes in the use case metamodel based on the activity tables. In the sequel, we give the detailed instantiation process.

- *sysName*. It is the system name.
- *SuD set*. First, we can retrieve the names of all SuDs from the activity tables, and all these names will be added to the SuD set. Second, we can retrieve the SuD sets for each SuD's subsystem. Note that the relations between subsystems can be determined by the scope of the use case.
- *User Set*. All the primary actors listed in the activity tables, except those included in the SuD set, constitute the User Set.
- *Use Case set*. *uName* can be constructed by *sudName* and Use Case in the table header, i.e. *sudName+"-"+use case*; *sName* is the SuD in the table header; *pActor* is the Primary Actor in the table header; *activities* is the set of all Activity-Names in the table body; *uActivitySet* can be constructed from the body part of Activity Table in the following:
  (i) The activity set is built according to the *activity-name, activity-type, comm-type, sender, receiver* in the body part of the Activity Table. No activity information is required for the *abort* activity and the internal activities.
  (ii) The activity sequence is the sequence containing all the SuD's activities in the use case. In the sequence, the activities are connected by the following operations:
    - ';' (sequential): denoting that the activities in Main scenario are in sequential relation;
    - \$or\$ (selection): describing the activities in Main scenario and their alternated activities in Variation and exceptional activities in Extension;

Algorithm 2: Configure Use Case Metamodel from Activity Tables

```
   Input: Activity tables
   Output: Use case model written in XML
1  text = readActivityTable();
2  Get the head of use case model getHeaderOfUCModel(text);
3  Get the SuD relations getSuDRelations(text);
4  Get the users getUsers();
5  pos = 0;
6  current_UC = getOneUC(text, pos), next_UC = getOneUC(text, current_UC);
7  while current_UC do
8      uName = getUseCaseName(); sName = getSuDName();
9      pActor = getPrimaryActor(); pre = getPreCondition(); post = getPostCondition();
10     store the header information of each use case;
11     Initialize two string variables: str1 ="", str2 ="";
12     for each activity description A do
13         aName = getActivityName(a), aType = getActivityType(a);
             cType = getCommunicationType(a); refer = isUCReference();
14         sender = getSender(a); receiver = getReceiver(a); cond = getCondition();
15         aci =< "activities aName =" +aName+ "aType = " +aType+ "cType=" +cType+ "sender="
             +sender+ "receiver=" +receiver+ "/ > \n";
16         str1+ = aci;
17         if A in main scenario then
18             if SerialNumber = 1 then
19                 str2+ = aName
20             else if there are related activities in variation or extension then
21                 str2 = str1+ ";" +aName;
22                 while getRelatedActivityName() do
23                     RA = getRelatedActivityName();
24                     if RA contains stepj then
25                         form a loop between A and activity j(A)
26                     else
27                         str2 = str2+ ";(skip$or$" +RA+ ")"

28     output str2 and str1;
29     output ending information of a use case;
30     current_UC = getOneUC(text, next_UC), next_UC = getOneUC(text, current_UC);
```

- b*() (loop). If under condition b, j^{th} step (*activity j*) can be back to i^{th} step (*activity i*), $i < j$, then all the activities between them are in a loop relation.

If isUCReference is true for some activity, then find the referred use case using the activity name, and replace it with the activity sequence of the referred use case.

Algorithm 2 shows the procedure to generate the use case source model by configuring the use case metamodel. For example, Table 5.1 can configure the activity class and generate an activity instance, as shown in Fig. 5.5, contained in the corresponding use case model. Applying Algorithms 1 and 2 to the textual use cases shown in Figure 5.2, we can get the corresponding use case model, which is shown in Figure 5.6.

:activity
aName=sendsItem aType=P cType=asyn condition=null isUCReference=false sender=Seller receiver=MIS

FIGURE 5.5: The activity class in the use case model based on the activity table given in Table 5.1.

5.2 Configuring the Component Metamodel: Mapping Use Case Models to Graphical SCA Models

By applying the transformation rules to the source model (that is, by running transformation rules specified in Figure 4.2 with the source model described in Figure 5.6 as input), we obtain the target model (or the configuration of component metamodel), as shown in Figure 5.7.

The following are the rules to map the target model to an equivalent SCA model.

- Each component of the target model is mapped to a SCA component;
- The *pp/pr* of a target component will generate *provided/required* port the SCA component, respectively. For example, a text description (in *pp*) in target model is *'P-sendsItem-asyn'*, then the corresponding port description is $(\overline{sendsItem}, \mathsf{provided}, \mathsf{asyn})$.
- The elements in *innerComs* of a target component become the sub-components of an SCA component.
- The *wireSet* of a target component is mapped to a wire set of the SCA component. For example, a wire description before mapping is 'submitsInformation:CL->CS', after mapping, it becomes $\{P^2_{submitsInformation} \longrightarrow P^1_{submitsInformation}\}$, where $P^2_{submitsInformation}$ is a port activity of component CL and $P^1_{submitsInformation}$ is a port activity of component CS.

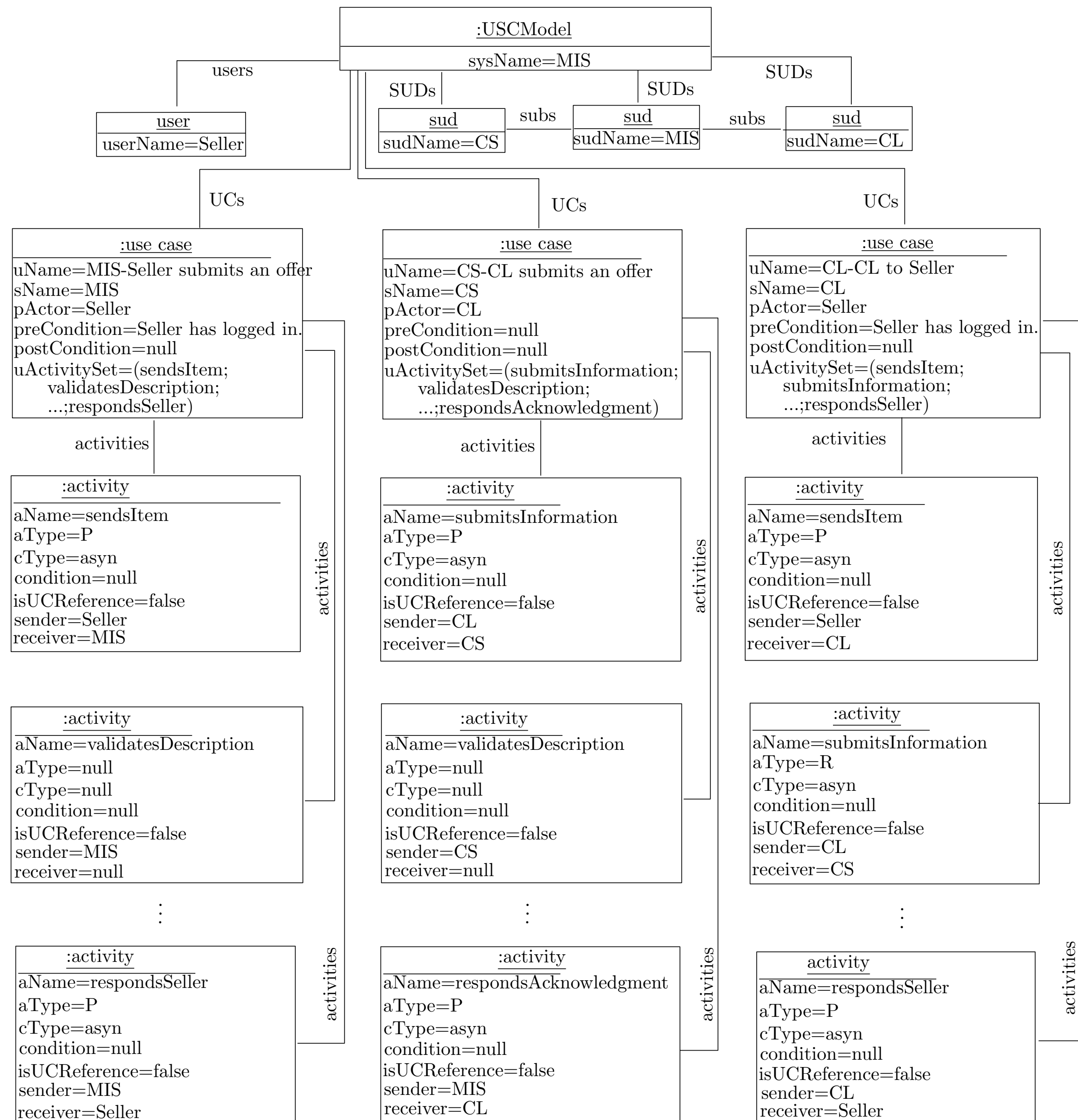


FIGURE 5.6: The source model of MIS (three use cases) during model transformation.

- The value of the behavior of a target component is mapped to a behavior expression for an SCA component. The activities in the behavior is replaced by corresponding port activities, and the text connectors (such as \$or\$,\$and-or\$) are replaced by formal connectors ($\lhd b \rhd$, $\|$).

Based on the above rules, we obtain a SCA model of MIS (the one as shown in Figure 3.5). While the formalized representation can be used for the analysis, the informal representation provides a good picture for designers to understand the system.

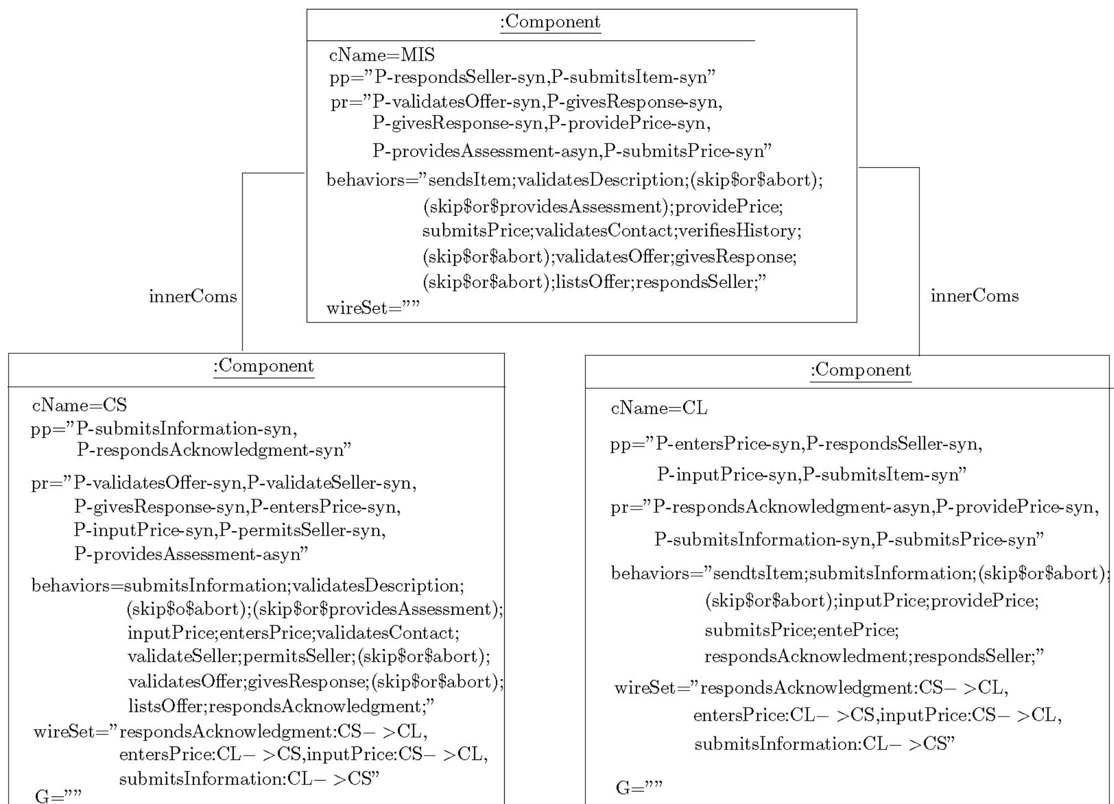


FIGURE 5.7: The target model of MIS (three components) generated from model transformation.

Chapter 6

Experiments

In this chapter, we demonstrate the effectiveness and efficiency of the proposed approach. We have implemented our approach with a prototype, which is a plugged-in software of the Eclipse. We will first give a brief introduction of the prototype and then give the detailed experimental results.

6.1 Description Of the Prototype

The implemented prototype consists of four parts: the textual use case editor, the source model generator, the model transformer, and the SCA model displayer. The textual use case provides an interface for users to write or import textual use cases and further conduct grammar checking and sentence style checking on these use cases. The source model generator applies the Stanford Parser to analyse textual use cases, the resulting information is used to build activity tables and then to generate the use case models. The model transformer accepts the use case model provided by the source model generator as input, and then runs the model transformation program to produce the component model. At last, the SCA model displayer extracts information from the component model to draw a SCA model. For the prototype, the use case editor, the source model generator and the SCA model displayer are implemented in Java, but the model transformer is implemented in ATL.

6.2 Experiments

We have conducted more experiments using our prototype. The examples are taken from the existing literature and the website. These examples cover Car Instrument Cluster System (CICS) [8], Elevator System (ES) [49], Nighttime Bank Deposit System (NBDS) [39], Supermarket Checkout System (SCS) [50], and ATM [1]. We have asked the company ”Zhejiang Innovation Central Software Co. LTD”[2] to manually build the SCA models. These examples took 2 experienced team leaders 6 hours to complete the SCA diagrams. For the justification, all these five examples have been implemented by their teams based on their design models. The systems have been tested to satisfy the requirements. We use *Comp(#)* to represent the number of components, *Sub-comp(#)* to represent the number of the subcomponents, *PP(#)* to represent the number of provided ports, *PR(#)* to represent the number of required ports, and *Rel(#)* to represent the number of relations in the wireset. Table 6.1 shows the statistics data produced by two team leaders.

TABLE 6.1: SCA statistics data produced by two team leaders.

Systems	Comp(#)	Sub-comp(#)	PP(#)	PR(#)	Rel(#)
CICS	3	2	6	4	5
ES	3	2	10	12	12
NBDS	6	5	15	14	15
SCS	4	3	12	11	14
ATM	3	2	13	11	12

Meanwhile, we used our prototype to design the SCA models. Table 6.2 shows the statistics data produced by our prototype and Table 6.3 shows the ’correct’ data (assume that the data from industry is correct) produced by our prototype.

To measure the accuracy of our method, we compute the *Precision* and *Recall* based on the following formulas:

$$\text{Precision} = \frac{\text{N}}{\text{M}} \times 100\%,$$

[1]http://www.math-cs.gordon.edu/courses/cs320/ATM
[2]www.centralsoft.com.cn

TABLE 6.2: SCA statistics data produced by our prototype.

Systems	Comp(#)	Sub-comp(#)	PP(#)	PR(#)	Rel(#)
CICS	3	2	6	4	5
ES	3	2	10	12	10
NBDS	6	5	14	14	12
SCS	4	3	11	10	13
ATM	3	2	15	11	12

TABLE 6.3: The correct SCA statistics data produced by our prototype.

Systems	Comp(#)	Sub-comp(#)	PP(#)	PR(#)	Rel(#)
CICS	3	2	6	4	5
ES	3	2	9	12	8
NBDS	6	5	13	14	11
SCS	4	3	10	9	12
ATM	3	2	13	11	10

TABLE 6.4: Accuracy Analysis of our prototype

Single System	Precision	Recall	All Systems	Precision	Recall
CICS	100%	100%	Components	100 %	100%
ES	91.9%	87.2%	Sub-components	100 %	100%
NBDS	96.1%	89.1%	PP	91.1%	91.1%
SCS	92.7%	86.4%	PR	98%	96.2%
ATM	90.7%	95.1%	Relations	88.5%	79.3%

$$\text{Recall} = \frac{\text{N}}{\text{S}} \times 100\%,$$

where N is the number of correct parts of the data produced by our method, M is the number of data produced by leaders, and S denotes the complete number of the correct data, that is, the data of the SCA model produced by the team leaders. *Precision* shows the effectiveness of discovering the design parameters of SCA that are meaningful conceptually in the application domain, and *Recall* measures the coverage of the correct parts of the generated model to the whole correct model. Table 6.4 shows the *Precision* and *Recall* for each system and different indexes of overall systems.

Table 6.4 shows that our prototype has a quite high accuracy for each system. In our experiments, the generated SCA component model of CICS is the same as that constructed by team leaders. For system ES, its *Precision* and *Recall* are 91.9% and 87.2%: one port description is not matched with the correct one, which leads to two

wrong wire descriptions, also there are another two wires not generated. For NBDS, its *Precision* and *Recall* is 96.1% and 89.1%: one port is omitted and one port is not described in the right way, as a result, one wire description is wrong and another three wires are omitted. For SCS, its *Precision* and *Recall* are 92.7% and 86.4%: two ports are omitted and one port is not described properly, thus the wires between them are omitted and a wrong wire is generated. For ATM, the *Precision* and *Recall* of ATM are 90.7% and 95.1%: two additional ports are generated and this introduces two wrong wire descriptions.

Also, Table 6.4 shows that our prototype works quite well in the different levels of the component model. Firstly, the *Precision* and *Recall* in the component and sub-component model are 100%, which shows that our prototype can capture the global signature of the SCA component model successfully. Secondly, the *Precision* and *Recall* of the port level are all higher than 90%, which indicates that our prototype is capable of generating most of the ports correctly although there are some mistakes in the port level. The main reason for the inaccuracy in the port level is that the ambiguous description of the textual use cases leads to misunderstanding of the activity, which leads to the wrong ports. There are two kinds of mistakes for ports: 1) The number of ports is wrong. If the NLP treats an interactive activity as an internal activity, then it is possible that a port of a component may be omitted; or if the NLP treats an internal activity as an interactive activity, then a port may be added to the component. 2) The information of a port is inconsistent. The algorithm to extract the *Activity-Type* and *Comm-Type* from an activity is sensitive such that small changes will impact the analysis process. Also, if the participants are not specified clearly in an activity description, then the description of the *Activity-type*, *Sender* and *Receiver* will contain inconsistent data. The *Precision* and *Recall* of the wire set level are much lower than the previous two levels. The main reason is that the wire set of the component is related to both the port description and port relations. However, it is apparent that our prototype can correctly generate at least one-half wires for each system.

6.3 Threat to Validity

The above experimental data and analysis show that our method is quite effective in transforming the use case model to SCA component model, while there are some threats to the evaluation of the effectiveness of our method.

Firstly, the completeness and precision of the textual use cases are crucial. Because the extraction of the use case model relies on the NLP technique, the constructed SCA model would be inaccuracy if the initial textual use cases are not well configured. It is worthwhile to investigate the characteristics of the commonly used software requirements which are specified by textual use cases to find a more flexible way to extract the use case model.

The second threat is the generalization of our method. In this paper, only some small examples, such as MIS, ATM, etc, are studied and analysed. They are quite simple examples of service composition. Hence, we should apply our method to some more complicated service-based software.

The third threat is the correctness of our transformation tool. Our transformation tool includes the analysis of the textual use cases, the generation of the use case model, the transformation from the use case model to the component model, and the graphic display of the SCA component model. The tool is tested to guarantee its correctness, and each sub-part of the tools is tested individually, then, the whole tool is tested.

The further threat is the evaluation of the accuracy of the result component model. If developers of the transformation tools also participate in the experimentation, it will cause some inaccuracies.

Chapter 7

Conclusion

In this thesis, we proposed a metamodel-driven method to transform textual use cases to service component models automatically. We first extracted the common features of use cases and service components, based on which we proposed the metamodels of use case models and component models. Then, we proposed the uniform transformation rules that can map use case models to component models based on the definitions of metamodels; moreover, we give a rigorous proof of the correctness of the proposed rules. The experimental results show that our method can be used to transform CIMs into PIMs more efficiently.

In the future, we will improve the proposed method in the following directions.

- We will support more sentence structure types in our sentence base. Currently, the sentence base contains six types of sentence structures. However, it cannot describe all kinds of requirements, such as those described by model verbs and fuzzy words. For example, the current sentence structure types cannot deal with the requirements “The application may retry the connection up to three times under unstable network conditions”, where the sentence contains the model verb “may”, and “The system should respond within roughly 5 seconds under normal conditions”, which contain a modal verb “should” and a fuzzy word “roughly”. They cannot be addressed by current sentence base. Note that modal verbs and

fuzzy words are common in specifications where flexibility is essential. Therefore, in the future, we aim to increase the support of sentence types to write more requirements. For example, we plan to incorporate new components to handle modal verbs and fuzzy terms, associating fuzzy numbers with appropriate membership functions to accurately interpret these terms.

- We will investigate the application of large language models (LLM) to configure the use case metamodel from textual requirements. Currently, natural language processing (NLP) techniques are used to extract information from textual use cases. Then the retrieved information is used to instantiate the attributes of the use case metamodel. Due to the nature of NLP techniques, some information may be lost. Therefore, more advanced techniques may be required to extract more information to generate richer component models. With the rapid development of LLM techniques, in the future, we can apply LLM to help the instantiation of the use case metamodel and the transformation.

- We will provide more types of use case metamodels to support more PIMs. The use case metamodel will determine the use case model, which in turn will determine the system design model. Therefore, the type of use case metamodel plays a critical role in generating the formal design model. In this thesis, since we focus on component-based software system design, we put some expert design knowledge into the use case metamodels. However, the use cases themselves do not explicitly display any system design information. Therefore, in the future, we can consider more types of use case metamodels to support more types of system design, such as object-oriented design.

Bibliography

[1] Z. Ding, M. Jiang, and J. Palsberg, "From textual use cases to service component models," in *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, 2011, pp. 8–14.

[2] Object Management Group, "OMG MDA Guide rev. 2.0." 2021. [Online]. Available: file:///Users/yzhou/Downloads/ormsc-14-06-01.pdf

[3] A. León, M. Y. Santos, A. García, J. C. Casamayor, and O. Pastor, "Model-to-model transformation: From UML class diagrams to labeled property graphs," *Business & Information Systems Engineering*, vol. 66, no. 1, pp. 85–110, 2024.

[4] B. Kim, L. Feng, O. Sokolsky, and I. Lee, "Platform-specific code generation from platform-independent timed models," in *2015 IEEE Real-Time Systems Symposium*. IEEE, 2015, pp. 75–86.

[5] M. Melouk, Y. Rhazali, and H. Youssef, "An approach for transforming CIM to PIM up To PSM in MDA," *Business & Information Systems Engineering*, vol. 170, pp. 869–874, 2020.

[6] T. D. Breaux, A. I. Antón, and J. Doyle, "Semantic parameterization: A process for modeling domain descriptions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 2, pp. 1–27, 2008.

[7] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 3, pp. 277–330, 2005.

[8] L. Kof, “Scenarios: Identifying missing objects and actions by means of computational linguistics,” in *15th IEEE International Requirements Engineering Conference (RE 2007)*. IEEE, 2007, pp. 121–130.

[9] Z. Ding, M. Jiang, and M. Zhou, “Generating petri net-based behavioral models from textual use cases and application in railway networks,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 12, pp. 3330–3343, 2016.

[10] G. Abowd, R. Allen, and D. Garlan, “Using style to give meaning to software architecture,” in *Proc. of SIGSOFT ‘93: Foundations of Software Engineering*, 1993, pp. 9–20.

[11] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.-V. Chioasca, and R. T. Batista-Navarro, “Natural language processing for requirements engineering: A systematic mapping study,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–41, 2021.

[12] The Stanford Natural Language Processing Group, “Stanford Parser,” https://nlp.stanford.edu/software/lex-parser.html#About.

[13] S. Sekine and R. Grishman, “Apple Pie Parser,” https://nlp.cs.nyu.edu/app/.

[14] SIL LANGUAGE TECHNOLOGY, “PC-PATR,” https://software.sil.org/pc-patr/.

[15] XTAG Research Group, “A lexicalized tree adjoining grammar for english,” IRCS, University of Pennsylvania, Tech. Rep. IRCS-01-03, 2001.

[16] “CHILL,” https://www.cs.utexas.edu/~ml/chill.html.

[17] H. W. Nissen, M. A. Jeusfeld, M. Jarke, G. V. Zemanek, and H. Huber, “Managing multiple requirements perspectives with metamodels,” *IEEE Software*, vol. 13, no. 2, pp. 37–48, 1996.

[18] OMG, “Omg unified modeling language specification, version 1.3,” 1999.

[19] T. Nakatani, T. Urai, S. Ohmura, and T. Tamai, “A requirements description metamodel for use cases,” in *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE, 2001, pp. 251–258.

[20] A. Durán, B. Bernárdez, M. Genero, and M. Piattini, "Empirically driven use case metamodel evolution," in *UML 2004—The Unified Modeling Language. Modeling Languages and Applications: 7th International Conference*. Lisbon, Portugal: Springer, 2004, pp. 1–11.

[21] A. Durán, A. Ruiz-Cortés, R. Corchuelo, and M. Toro, "Supporting requirements verification using xslt," in *Proceedings IEEE Joint International Conference on Requirements Engineering*. IEEE, 2002, pp. 165–172.

[22] A. Goknil, I. Kurtev, and K. van den Berg, "A metamodeling approach for reasoning about requirements," in *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings 4*. Springer, 2008, pp. 310–325.

[23] M. R. Dube, "Enhanced uml use case meta-model semantics from cognitive and utility perspectives," in *Advances in Computing and Data Sciences: 4th International Conference, ICACDS 2020, Valletta, Malta, April 24–25, 2020, Revised Selected Papers 4*. Springer, 2020, pp. 85–95.

[24] B. Hnatkowska and P. Zabawa, "A reusability-oriented use-case model specification language," in *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2023, pp. 567–576.

[25] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software & Systems Modeling*, vol. 18, pp. 2361–2397, 2019.

[26] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 419–430.

[27] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software & systems modeling*, vol. 15, pp. 647–684, 2016.

[28] D. Akehurst and S. Kent, "A relational approach to defining transformations in a metamodel," in *International Conference on the Unified Modeling Language*. Springer, 2002, pp. 243–258.

[29] D. Milicev, "Automatic model transformations using extended uml object diagrams in modeling environments," *IEEE Transactions on Software Engineering*, vol. 28, no. 4, pp. 413–431, 2002.

[30] Object Management Group, Inc. (OMG), "Query/view/transformation specification," 2016, accessed: 2024-10-20. [Online]. Available: https://www.omg.org/spec/QVT/1.3/PDF

[31] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "Viatra-visual automated transformations for formal verification and validation of uml models," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. IEEE, 2002, pp. 267–270.

[32] G. Engels, R. Heckel, and J. M. Küster, "Rule-based specification of behavioral consistency based on the uml meta-model," in *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference*. Toronto, Canada: Springer, 2001, pp. 272–286.

[33] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen, "Consistency-preserving model evolution through transformations," in *UML 2002—The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference*. Dresden, Germany: Springer, 2002, pp. 212–227.

[34] A. Schürr, "Specification of graph translators with triple graph grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1994, pp. 151–163.

[35] J. Whittle, "Transformations and software modeling languages: Automating transformations in uml," in *International Conference on the Unified Modeling Language*. Springer, 2002, pp. 227–242.

[36] N. Koch, G. Zhang, and M. J. Escalona, "Model transformations from requirements to web system design," in *Proceedings of the 6th international conference on Web engineering*, 2006, pp. 281–288.

[37] J. J. Gutiérrez, C. Nebut, M. J. Escalona, M. Mejías, and I. M. Ramos, "Visualization of use cases through automatically generated activity diagrams," in *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28-October 3, 2008. Proceedings 11*. Springer, 2008, pp. 83–96.

[38] D. Kulak and E. Guiney, *Use cases: requirements in context*. Addison-Wesley, 2012.

[39] A. Cockburn and L. Cockburn, *Writing effective use cases*. Pearson Education India, 2008.

[40] Z. Ding, Z. Chen, and J. Liu, "A rigorous model of service component architecture," *Electronic Notes in Theoretical Computer Science*, vol. 207, pp. 33–48, 2008.

[41] Z. Ding, M. Jiang, and A. Kandel, "Port-based reliability computing for service composition," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 422–436, 2011.

[42] A. Narayanan and G. Karsai, "Verifying model transformations by structural correspondence," *Electronic Communications of the EASST*, vol. 10, 2008.

[43] F. Plasil and V. Mencl, "Getting'whole picture'behavior in a use case model," *Journal of Integrated Design and Process Science*, vol. 7, no. 4, pp. 63–79, 2003.

[44] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 99–115, 2003.

[45] C. Damas, B. Lambeau, P. Dupont, and A. Van Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1056–1073, 2005.

[46] I. T. S. Sector, "Itu-t recommendation z. 120," *Message Sequence Charts (MSC96)*, 1996.

[47] A. Van Lamsweerde and L. Willemet, "Inferring declarative requirements specifications from operational scenarios," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1089–1114, 1998.

[48] R. Alur and M. Yannakakis, "Model checking of message sequence charts," in *International Conference on Concurrency Theory*. Springer, 1999, pp. 114–129.

[49] A. Shui, S. Mustafiz, and J. Kienzle, "Exception-aware requirements elicitation with use cases," *Advanced Topics in Exception Handling Techniques*, pp. 221–242, 2006.

[50] Y. Shinkawa, "Model checking for uml use cases," in *Software Engineering Research, Management and Applications*. Springer, 2008, pp. 233–246.