A Pattern-based Approach to Requirements Formalization and Its Supporting Tool

著者	WANG Xi
著者別名	王 晳
その他のタイトル	パターンに基づく要求仕様の形式化方法及び支援ツールに関する研究
page range	1-158
year	2014-03-24
学位授与番号	32675甲第333号
学位授与年月日	2014-03-24
学位名	博士(理学)
学位授与機関	法政大学 (Hosei University)
URL	http://hdl.handle.net/10114/8868

博士学位論文 2013 年度

パターンに基づく要求仕様の形式化方法及び支援ツールに関する研究

法政大学　審査学位論文

法政大学　情報科学研究科　情報科学専攻博士課程

王　　哲

パターンに基づく要求仕様の形式化方法及び支援ツールに関する研究

A Pattern-based Approach to Requirements Formalization and Its Supporting Tool

by

Wang Xi

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

Graduate School of Computer and Information Sciences

HOSEI UNIVERSITY

March 2014

Advisor: Professor Shaoying Liu, Hosei University

Contents

0 Preface **9**

1 Introduction **13**

1.1 Requirements in software engineering . . . 14

1.2 Formal methods and formal specification . . . 16

1.3 Motivation of the research . . . 18

1.4 Our solution — A pattern-based approach . . . 21

1.5 Summary . . . 25

2 Related Work **26**

2.1 Methods for supporting requirements formalization . . . 26

2.1.1 Methods for facilitating formal specification construction . . . 26

2.1.2 Methods for transforming informal requirements into formal specifications . . . 29

2.2 Tool support for requirements formalization . . . 30

2.3 Methods for supporting requirements clarification . . . 31

2.4 Methods for data type declaration . . . 32

2.5 Methods for pattern knowledge representation . . . 34

2.6 Summary . . . 36

3 An overview on the pattern-based approach to requirements formalization **37**

3.1 Preliminaries . . . 37

3.1.1 SOFL . . . 37

3.1.2 Pattern . . . 41

3.2 The outline of the approach . . . 42

3.3 An example system . . . 45

3.4 Summary . . . 47

4 Specification pattern system **48**

4.1 Specification pattern . . . 48

4.1.1 Pattern structure . . . 50

4.1.2 Pattern definition . . . 52

4.2 The hierarchy in the pattern system . . . 62

4.3 Summary . . . 67

5 Requirements formalization based on the specification pattern system **68**

5.1 From informal requirements to formal specifications . . . 68

5.2 Data type declaration . . . 76

5.2.1 Approach outline . . . 77

5.2.2 Function-related declaration . . . 79

5.2.3 Type combination . . . 81

5.2.4 Property-guided declaration . . . 84

5.2.5 Priority-guided declaration . . . 87

5.2.6 Expression update . . . 88

5.2.7 Case study . . . 89

5.3 Summary . . . 95

6 Representation of the pattern knowledge **96**

6.1 Attribute tree . . . 96

6.1.1 The definition of attribute tree . . . 97

6.1.2 requirement tree . . . 101

6.1.3 Type tree . . . 107

6.2 HFSM . . . 112

6.2.1 The definition of HFSM . . . 112

6.2.2 Representing the pattern knowledge in HFSM . . . 116

6.3 Summary . . . 121

7 Prototype tool for supporting the pattern-based approach **123**

7.1 Tool design . . . 123

7.1.1 Knowledge extractor . . . 124

7.1.2 Guidance generator . . . 127

7.1.3 Preprocessor . . . 127

7.2 Tool implementation . . . 127

7.3 Tool interface . . . 131

7.4 Case study . . . 133

7.5 Summary . . . 137

8 Experiment **139**

8.1 Experiment for investigating the ability and usability of the tool . . . 139

8.2 Experiment for evaluating the effectiveness of the tool . . . 141

8.3 Summary . . . 144

9 Conclusion and future work **145**

9.1 Conclusion . . . 145

9.2 Future work . . . 146

9.2.1 Future research on the pattern-based approach . . . 146

9.2.2 Future research on the supporting tool . . . 147

List of Figures

1.1 The six phases for software developement in software engineering 14

1.2 The underlying principle of formal methods 17

3.3 An example CDFD hierarchy 38

3.4 The module specification of the example CDFD hierarchy 40

3.5 The outline of the pattern-based approach 43

3.6 The CDFD of the banking system 46

4.7 The underlying theory of the pattern-based approach 49

4.8 Pattern "sorting" 51

4.9 The structure of clarification rule repository 58

4.10 The formal definition of the pattern "sorting" 60

4.11 The formal definition of the pattern "group" 63

4.12 Pattern categorization 64

5.13 Types and variables declared for the banking system 71

5.14 The formal representation of the function *transaction analysis* 75

5.15 The outline of the data type declaration approach 78

5.16 The main procedure of the property-guided declaration 85

5.17 The specification pattern *alter* 93

5.18 The piority-guided declaration process during the application of the pattern "alter" 94

6.19 The structure of attribute tree 97

6.20 The attribute tree of the pattern *alter* 100

6.21 The strcuture of requirement tree 102

6.22 The requirement tree of the example alter operation 104

6.23 The attribute trees of the pattern *add* and *delete* 106

6.24 Pseudo node for special situations 108

6.25 Pseudo node for special situations 108

6.26 Comparison between attribute tree and type tree 109

6.27 The type tree constructed for the example display function 113

6.28 The formal expression generated based on the example type tree 113

6.29 An example FSM model 115

6.30 Example HFSM models 116

6.31 The top level FSM in the HFSM 118

6.32 The HFSM describing the pattern selection process 119

6.33 The FSM for describing the application of individual patterns 120

6.34 The FSM describing the application of the pattern "sort" 121

7.35 The design of the tool for supporting requirements formalization 124

7.36 The algorithm for utilizing the HFSM 126

7.37 An example FSM and its XML representation 130

7.38 The main frame of the tool 131

7.39 The snapshot of the supporting tool 133

7.40 The defined types and variables in the module $systemModule$ 134

7.41 The snapshot of the supporting tool 135

7.42 The snapshot of the supporting tool 136

7.43 The snapshot of the supporting tool 137

List of Tables

3.1 The constructors of the compound types in SOFL 41

4.2 An overview on basic element types 54

5.3 Solution table for type combination 83

5.4 Pattern "belongTo" 91

5.5 Priority in Pattern "belongTo" 92

7.8 Guidance produced from the knowledge retrieved by the *knowledge extractor* 128

7.9 Tags for identifying the elements in the FSM model 129

7.10 Symbols involved in FSM models 130

8.11 The result of the first experiment 141

8.12 The result of the second experiment 142

ACKNOWLEDGMENTS

I want to give my sincere thanks to my supervisor Professor Shaoying Liu for his great support throughout my PHD study. As a supervisor in academics, he often discusses with the students about how to do research and shares his own experience which will benefit our future research work. He pays attention to the research process of each student and provides insightful suggestions to the research work. Seminars are frequently held to find out and help overcome the difficulties we encountered, although he is busy at work. He is very patient in revising our paper manuscripts and even gives comments on the detailed mistakes and inappropriateness in the paper. As a supervisor in life management, he always stimulates our enthusiasm in both research and life through his sense of humor and encourages us whenever we are frustrated. He understands our stress and emphasizes the balance between working and exercises. He took us to sports many times for a relief from the busy study.

I appreciate the professors and colleagues who have provided me with kind help for my TA work. They have been very supportive when I was a beginner in TA. I appreciate the colleagues in the lab for their stimulation in new ideas and assistance in both research and daily life. I also appreciate the students participating in my experiment for their cooperation.

I thank the staffs in the administration office and the sports room management office. They take my issues as their own and are always patient when providing services.

I appreciate the teachers and the staffs in the Japanese classroom. They treat each student as their own child and always care about our lives in Japan. They often organize various activities to help us get to know the Japanese culture and better join the society.

I thank my family for their understanding and support.

ABSTRACT

Despite the effectiveness of requirements formalization in producing accurate requirements documentation and deepening the developers' understanding of the envisioned systems, this technique can hardly be accepted by software industry mainly because it requires mathematical sophistication and considerable experience in using formal notations, which remains a challenge to many practitioners. Many methods and tools have been proposed to deal with the problem by providing general guidance or automatic support in transforming informal requirements into formal specifications. However, they fail to accomplish the task when encountering incompleteness and ambiguities in the informal requirements.

To handle this challenge, this thesis describes a *pattern-based approach* to facilitating the formalization of requirements. In this approach, a *specification pattern system* is pre-defined to guide requirements formalization where each pattern provides a specific solution for formalizing one kind of function into a formal specification. All of the patterns are classified and organized into a hierarchical structure according to the functions they can be used for formalization. The distinct characteristic of our approach is that all of the patterns are stored on computer as knowledge for creating effective guidance to facilitate the developer in requirements formalization; they are "understood" only by the computer but transparent to the developer. Based on the pattern system, a method that guides the requirements formalization process by applying the pattern system is described. To facilitate the understanding of the guidance produced by the pattern system and the utilization and maintenance of the pattern knowledge, a method for representing the pattern system is proposed where attribute tree and HFSM are adopted. These two notations are used to represent different parts of the pattern knowledge. The method for applying the pattern knowledge represented in the two notations is given.

We also describe a prototype tool that supports the pattern-based approach. The tool derives necessary functional details of the intended requirement through interactions with the developer and generates a formal specification according to the obtained information. Two experiments on the tool supported approach are presented to demonstrate the effectiveness of the approach.

Preface

Our software-dependent life and society require an effective method for developing high-quality software, since even a tiny bug would drag us into a catastrophe where we have to pay considerable prices.

Many researches have been done in software engineering to enhance software quality and the introduction of formal methods is regarded as a milestone. This technique integrates mathematics with software development and takes advantage of formalism. It mainly involves two parts: formal specification and formal verification. Formal specification documents software behaviors in formal notations, which avoids ambiguities in requirements specification and misunderstanding among the members of the software development team. It also allows rigorous analysis and automatic manipulations. Formal verification verifies the consistency between formal specification and its implementation alternatives to ensure that the resultant program performs the expected behavior.

However, a long distance still exists from the fundamental theory and the real practice in industry. We have attended several conferences on formal methods and met many practitioners from software companies. Most of them are satisfactory with the power of formal methods but doubt its acceptance in industry since writing formal specifications is a huge challenge for them, not need to mention formal verification. Without addressing this challenge, formal methods would not play their expected role in practice.

The problem in writing formal specifications mainly stems from the complexity of the formal notations. If we can assist practitioners to obtain the target formal specifications without the need of dealing with formal notations manually, they will probably accept this easy way to high-quality software development technique.

We have studied the literature to understand why the existing approaches are not able to solve the problem. There are mainly two kinds of approaches for supporting requirements formalization. One is to provide guidance for writing formal specifications. The problem is that the guidance provided by most of these methods only involves a general solution to a general problem. Developers who want to use the solution need to understand it first. Then they need to analyze the detailed information of the real problem and instantiate the general solution as a specific solution to the problem. This process requires the understanding of the involved languages, which means that the developers still have to struggle with formal notations. The other kind of approach is to allow the description of software functions in informal languages that are easier to use, such as natural language and graphical notations, and automatically translate the informal languages into formal notations. Compared with the first kind, these approaches prevent developers from directly using formal notations. However, most of them only involve a set of syntactical rules without semantic support. They cannot guarantee the completeness of the informal requirement written by the developer and incomplete information would lead to erroneous formal specifications or even obstacle the translation process.

To this end, we introduce our *specification pattern* as a solution. A pattern treats a function as a composition of its attributes. Each pattern provides a framework for formalizing one kind of function where the relevant attributes are guided to be clarified and a formal specification is generated according to the clarified attributes. Specifically, the major contributions of the thesis are listed below.

1. *A specification pattern system for supporting requirements formalization*

   We design a specification pattern system as knowledge for guiding the requirements formalization process. The novelty of the system is that the involved specification patterns can be applied by machines for producing guidance and the users are unaware of their existence. These patterns work on the semantic level to guide requirements clarification and provide methods to transform the clarified requirements into formal specifications. The users are only required to follow the guidance and the patterns will handle the remaining tasks.

2. *The application method of the specification pattern system*

   In our approach, requirements formalization is actually the application process of the pattern system. Since the design of the pattern system considers the need of requirements formalization, the application of the system is straightforward. It includes pattern selection and pattern application. The former is aimed at giving the outline of the intended function while the latter further clarifies the included details until reaching a formal expression.

3. *An approach to representing the pattern knowledge*

   Based on the formal definition of the specification pattern system, we propose an approach to representing the included pattern knowledge in attribute tree and HFSM (Hierarchical Finite State Machine). The goal of the approach is to facilitate the understanding of the guidance produced by the specification patterns and the *utilization and maintenance of the pattern knowledge. We analyze the problem of using pattern structure or formal definition as the representation of the pattern knowledge and compare the two different representations to illustrate the necessity of the representation approach.*

4. *A prototype tool for supporting the pattern-based approach*

   To validate and evaluate the pattern-based approach, we developed a supporting tool and use it in the requirements formalization of experimental software projects. The tool interacts with the developers and displays the generated formal specification with the obtained information.

The thesis is organized as follows.

Chapter 1 describes the motivation of the research by first introducing the background and basic concepts and then showing the encountered problems to be solved. Then we briefly present our solution to these problems.

Chapter 2 compares our proposed approach with other related research results from different aspects. The comparison shows how our approach solves the problems better than others.

Chapter 3 first introduces two basic concepts SOFL and pattern. The former is the specific language that we adopt in this thesis to illustrate each technique of the approach and the latter indicates the source of the underlying theory of our approach. Then we present the framework of our approach and the relations between each involved components. Finally we introduce an example system which will be used to explicitly explain how each technique works in the following chapters.

Chapter 4 describes the *specification pattern system* which is the foundation of the pattern-based approach. *Specification pattern* is first introduced by explaining its structure and presenting its formal definition. Then the pattern system is introduced by describing how the patterns are organized into the architecture of the pattern system. A formal definition of the pattern system is also given.

Chapter 5 presents the method for applying the *specification pattern system* to guide the requirements formalization process. There are two activities to be interleavingly carried out in this method: describing informal requirements informal expressions and data type declaration. We describe these two activities in detail respectively and use example functions to illustrate them.

Chapter 6 proposes an approach to representing the pattern knowledge involved in the *specification pattern* *system.* Two languages attribute tree and HFSM are used in the approach to represent different pattern knowledge. We present each of their definitions in detail respectively and show how they are used to represent the pattern knowledge with examples.

Chapter 7 describes the prototype tool that we implemented to support the pattern-based approach. We first give the design of the tool that shows how the involved components cooperate to perform the required functionality. Then the detail of each component is explicitly presented. We also explain some implementation details of the tool and demonstrate its functionality through its interface and execution in formalizing example functions.

Chapter 8 presents two experiments on the prototype tool. The first experiment aims at exploring the domain that the tool can be applied and checking the validity of the pattern-based approach. The goal of the second experiment is to evaluate the effectiveness of the tool in facilitating requirements formalization by comparing manual formalization and tool-supported formalization.

Chapter 9 gives a conclusion on the research results in this thesis and summarizes the problems we found in developing our approach and the work that we need to do in the future to solve them and improve the approach.

Chapter 1

Introduction

Software has become a necessity for proper functioning of the society. Everyday millions of people around the world are transferred to their destinations by subway and aircraft controlled by software. Medical devices contain tens of thousands of lines of code to provide life-sustaining functions for patients. A large portion of the world financial activities depend on software, such as E-commerce, online-trading and ATM (Automatic Teller Machine) software. Smart phone works on software to connect billions of people with the world and provide them with various kinds of daily services.

While bringing significant efficiency and convenience, software would also lead us to great loses or even catastrophe if containing bugs. As more and more social functions are performed by computers, we may suffer from more serious consequence of software malfunctions. In March 2001, it was determined that 28 cancer patients were given radiation doses exceeding their required dosage at Panama's National Oncology Institute. By August 2005, 23 of these 28 patients died. It was reported that at least 18 of the patient deaths were attributed directly to radiation overdose. The major reason is the lack of warning mechanism in the software program for handling the improper usage [1]. In 2003, computer system failures cause 814 blackout in North America [2]. Phobos-Grunt, Russia's most ambitious planetary mission in decades, was launched on November 9, 2011. But it was lost due to a programming error and fell back to Earth on January 15, 2012 [3]. An update on the CA-7 software of The Royal Bank of Scotland Group was corrupted on June 2012 [4]. Large amount of customers' wages, payments and other transactions were disrupted.

Safety-critical application domains - including nuclear, defense, space, medical, and transport industries - increasingly depend on or are controlled by computer software [5] [6] [7]. On the other hand, as the complexities of the systems under control rise, software development becomes a complicated process. How to develop reliable

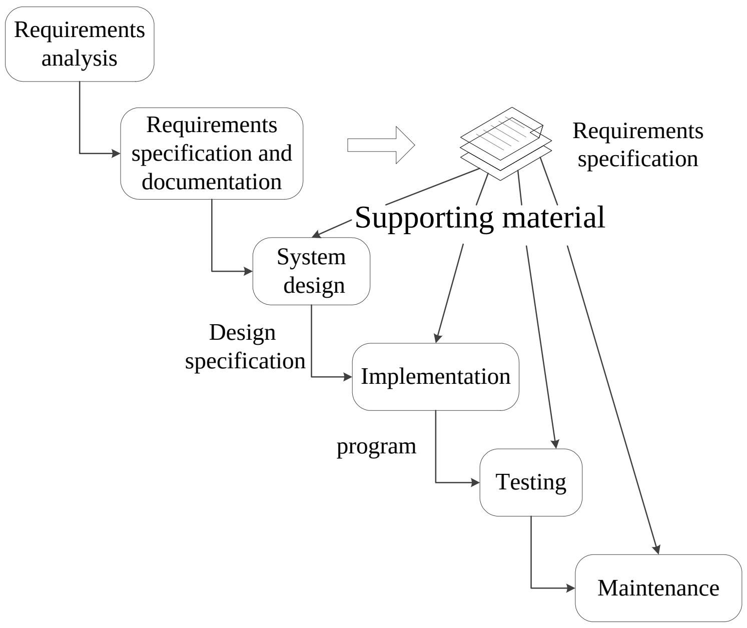


Figure 1.1. The six phases for software developement in software engineering

large-scale software in a systematic and efficient way is regarded as a critical and urgent problem to be solved in software engineering.

1.1 Requirements in software engineering

Software engineering is the study and application of systematic, disciplined, quantifiable approaches to the development, operation, and maintenance of software [8]. As a revolution to software development. The birth of software engineering changed the view on software: software is not only a program but a combination of documentation and program [9]. Therefore, the production of software is not just programming but a process comprising six phases as shown in Figure 1.1: *requirements analysis*, *requirements specification and documentation*, *system design*, *implementation*, *testing* and *maintenance*.

The *requirements analysis* phase studies the envisioned system and explores the necessary requirements that the system should satisfy. It solves the problem of "What the system needs to do?" and requires the collaboration of different roles in the development team including stakeholder, analyst and domain experts. The *requirements specification and documentation* phase collects and specifies the explored requirements in a specific language. It results in a requirements specification that describes the expected behaviors of the system under construction.

Based on the requirements specification, the *system design* phase produces a design specification for achieving a system that satisfies the requirements specification. In the design specification, the architecture of the entire system is given and the involved components are explicitly defined. The relations between these components are specified by defining the interface of each component and the involved data types. During the *implementation* phase, detailed algorithms are designed and a program is produced by implementing the design specification and the algorithms. Requirements specification also serves as a supporting material in this phase to ensure that the implementation details can satisfy the original requirements. In order to check whether the behavior of the implemented software satisfies the defined requirements, *testing* is carried out by running the produced program with well-defined test cases. It mainly comprises three steps: test case generation, program execution with the test cases and test results analysis. After the software faults detected in the *testing* phase are fixed, the software can be delivered to the customers. The *maintenance* phase updates the delivered software according to the requirements specification when errors are found during operation or new requirements is proposed.

It can be seen from the explanation of the above phases, well-written requirements specifications are the key to successful software projects [10]. They provide a clear direction for the development activity that may become a lengthy and error-prone process if being carried out from scratch. They also enable the identification of software flaws in the early stage of the development process, which costs much less than discovering design errors in the later stage. Furthermore, requirements specification serves as benchmark for evaluating and improving the quality of the produced software.

To achieve well-written requirements specification, we should first clarify what is a well-written requirements specification. According to IEEE guide to software requirements specifications, a good requirements specification should have the following properties [11].

- **Unambiguous** A requirements specification is unambiguous if and only if each included requirement has only one interpretation.
- **Complete** A requirements specification is complete if (1) it includes all significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces. (2) it defines responses to all realizable classes of input data in all realizable classes of situations. (3) it conforms to any

requirements specification standard that applies to it. (4) it gives full label and reference of all the included figures, tables, and diagrams, and defines all terms and units of measure.

- **Verifiable** A requirements specification is verifiable if and only if each included requirement is verifiable.
- **Consistent** A requirements specification is consistent if and only if no set of individual requirements described in it conflict.
- **Modifiable** A requirements specification is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently.
- **Traceable** A requirements specification is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement.
- **Usable during the Operation and Maintenance Phase** A requirements specification must address the needs of the operation and maintenance phase, including the eventual replacement of the software.

In the early days, practitioners adopt natural language to write requirements specifications since it is easy to use. Graphical notation and structure are later introduced to facilitate understanding and communications between team members. However, requirements specifications written in natural language or graphical notation can hardly meet the demand of the above properties due to the inevitable ambiguities. Without accurate definition, requirements specification would probably be incomplete, unverifiable, inconsistent, too disorganized to be modifiable, untraceable and difficult to be used in operation and maintenance phase. To deal with the problem, formal specification technique is proposed.

1.2 Formal methods and formal specification

Before presenting the concept of formal specification, formal methods is first introduced since it reveals the role of formal specification in software development.

Formal methods has made significant contributions to software engineering by establishing relatively mature techniques to formal specification, refinement, and verification, and its theoretical influence on conventional software engineering has been well known. The notion of pre- and post-conditions have been introduced into some

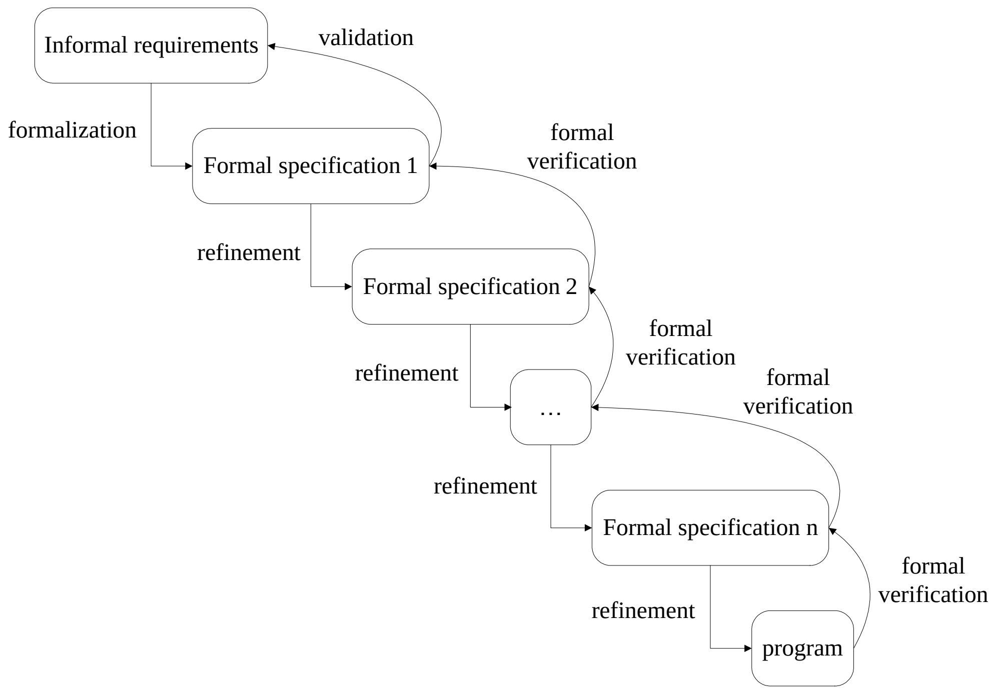


Figure 1.2. The underlying principle of formal methods

programming languages to support the "design by contract" principle [12] [13]; many companies have tried or actually used some formal methods in their real software projects [14]; and many practitioners have become more interested in formal methods nowadays than before (e.g., in Japan).

The definition of formal methods is the use of mathematical approaches in the specification, design, analysis and assurance of computer systems and software [15]. Its underlying principle is shown in Figure 1.2.

The first step in software development using formal methods is to formalize the customer's informal requirements into a formal specification. To check whether the formal specification is a faithful representation of the original informal requirements, various validation approaches are proposed such as animation [16] [17] [18]. If inconsistencies are found, the formal specification will be modified until being approved by the validation approaches. Then a formal refinement process is started where the formal specification is gradually refined into an executable program. For each step of the refinement, formal verification needs to be performed to ensure the consistency between the refined specification, or the final program, and its predecessor.

As one of the critical techniques in formal methods, formal specification describes expected system behaviors in mathematically based notations. With well-established formalism, it enables rigorous analysis of requirements and can be manipulated automatically. It also sets a firm foundation for the later stages of software development process and serves as prerequisite for verifying the correctness of the implementation alternatives using formal proof or specification-based testing [19]. There are many formal specification languages proposed in the literature, such as VDM (Vienna Development Method) [20], Z [21] and B [22].

Writing formal specifications is a requirements formalization process with two steps: clarifying the necessary functional details of the requirements and representing the clarified requirements in formal notations. This process helps deepen the understanding of the envisioned system and significantly improves the preciseness of the original requirements.

1.3 Motivation of the research

Many researches on applying formal methods to software engineering over the last twenty years have suggested that the formal specification technique can effectively help developers understand user requirements and system components (e.g., data items, operations) if it is applied appropriately [14] [23] [24]. However, "applied appropriately" in real practice remains a challenge. In spite of the statistic data that shows the improvement of software quality by using formal specifications [25] [26] [27] and the successful stories reported in the latest survey on industrial use of formal specification by Woodcock *et al.* [28], applications of formal specification to real projects in industrial are still rare [29]. Experience shows that industrial practitioners are interested in using formal specification technique to solve their problems occurring in exercising conventional software engineering techniques, but only a few of them with courage actually take actions [30, 31]; most of them turn away from the technique after they learn or try them for the first time. For example, in Japan several companies have tried Z and VDM (e.g., Nihon Unisys, Fujitsu, CSK, and FeliCa networks), but few of them have shown a positive sign toward the future use of the same method again.

The major reason is that the practitioners find it hard to express their ideas properly in formal notations. It is often the case that the writer of a formal specification, who can be an analyst or designer, understands what he or she wants to say in mind (or in natural language), but does not know exactly what formal expression can be used to properly express his or her informal idea. This may sound difficult to accept by a formal method researcher or a person with strong mathematical background, but it is the reality in the software industry where vast majority of software developers, especially software analysts and designers, may not even receive systematic training in computer science but have to face pressures to produce qualified software systems within limited time and budget [32]. They are not satisfied with the current practice for poor quality and high cost, and have expected formal methods to be a magic solution. Perhaps for this reason, many of them have a high interest in formal methods, but even they may understand the potential benefits of successful application of formal specifications, few of them really take action to use them in practice, simply because "it is complicated". This complicity stems from the fact that requirements formalization requires high skills for abstraction and experience in manipulating the formal notation in which the specification is written [29] [33]. It involves decisions to be made by the specification writer in order to clarify ambiguities. The practitioners need training and practice to enhance such skills, but because of the economy pressure, many of them would stop continuing to learn the techniques in training courses and to use them in real projects [34]. Even if they manage to understand formal notations through long-term practice, formalizing complex functions is still error-prone and costly. They will find themselves trapped in tedious syntax details, rather than working on function designs, especially when the behaviors to be described are complicated.

Specifically, according to the two steps for writing formal specifications, there are three problems to be handled when formalizing a requirement.

- What function details of the requirement are needed for formalization?

  Formalizing requirements into formal specifications is also a refinement process from abstract ideas to precise descriptions, during which designers have to consider the intended functions from many perspectives and add sufficient details. Let's consider the informal requirement "update the datastore D of a banking system". One cannot write a formal specification with the available information of the given requirement and needs to provide necessary function details first, including the parts to be updated in D and the operations for updating these parts. These details are clarified by specifying lower-level details such as the parts of each part to be updated in D and the operations for updating them. Such a clarification process continues until reaching the bottom-level function details. Incomplete information at any level will lead to the failure of the later formalization.

  Since most developers are lack of sufficient skills and experience in identifying the function details necessary to be clarified for further formalization, the resultant formal specifications are likely to be incomplete or even erroneous. Although many methods have been proposed to support the identification of function details, requirements clarified by most of them are often too abstract to be formalized. Moreover, these methods provide few specific guidelines on the requirement details that need to be clarified to form complete formal specifications; the clarification process can be time-consuming and error-prone.

- How to achieve these function details?

  For requirements that are sufficiently simple, achieving the required function details is not a problem. For example, when formalizing a requirement that describes "belongTo" relation, one needs to specify the member involved in the relation and the object that the member belongs to. This task can be easily done by providing the formal expressions that represent the required two objects. For complex requirements, especially data-intensive requirements, however, achieving all function details at one time would be overwhelming and the given details would be difficult to comprehend. For example, if the previously mentioned data store D is designed with a complex structure and the required function details forms a hierarchy with many layers, one needs to spend considerable time to study the inner structure of the requirement before achieving the function details. Besides, the complexity of the requirement also makes the provided information unreliable and represented in a disordered manner.

  Current methods for solving this problem focus on the design of easy-to-use languages for representing function details, such as UML [35]. They regard the specification of the function details as intelligent work and do not provide any guidance on the achievement of these function details. In this case, the completeness and correctness of the provided function details are hard to be guaranteed. Moreover, the designed languages are often used to informally describe abstract functions and incapable of specifying bottom-level function details precisely.

- How to formally represent the function details?

  With clarified function details, the difficulty of writing the corresponding formal specification depends on the complexity of the given function details. For example, "John_Smith belongs to Hosei_University" is a clarified requirement where the member involved in the relation is specified as "John_Smith" and the object that the member belongs to is specified as "Hosei_University". How to represent it in an appropriate formal specification will depend on what we mean by John_Smith and Hosei_University? If John_Smith is treated as a person and Hosei_University is a set of persons, a membership expression (e.g., John_Smith in set Hosei_University in VDM [20]) can be an appropriate formal specification. But if the data structure of Hosei_University is declared as a set of faculties and each faculty is a composite type of several fields, for example, "teachers", "students" and "administrators", and each field is declared as a set of persons, the formal specification of the requirement will be more complex.

When the complexity of the clarified function details reaches a relatively high level, organizing large amount of function details in a formal manner without introducing mistakes or missing any information becomes extremely difficult. Consider the following requirement "Display all the transactions on July 3rd belonging to the accounts that contain more than 3 transaction records on July 3rd, more than 500 US dollars and more than 1000 Japanese Yen". All the necessary function details for formalization are already given in this requirement, but representing these details in an appropriate formal specification is still challenging.

Without effective solutions for addressing the above problems, a great progress in spreading formal specification technique in industry and introducing it to ordinary practitioners would be highly impossible.

1.4 Our solution — A pattern-based approach

Experience suggests that resolving ambiguities of requirements is a learning process that often requires the analyst to make decisions and formalizing requirements is a means to precisely understand them. We believe that an effective way to solve the three problems in requirements formalization is to take the approach in which ambiguities of informal requirements are gradually clarified while the corresponding formal specifications are automatically generated. If the necessary function details can be correctly retrieved, its formalization only involves changing its format, which can be more efficiently and reliably done by a software tool [36].

To this end, a *pattern-based approach* to refining informal requirements into formal specifications is proposed in the thesis. In this approach, a *specification pattern system* is pre-defined where each *specification pattern* provides a specific solution for formalizing one kind of function. To facilitate pattern selection, all of the patterns are categorized into a hierarchy according to the functions they can be used to formalize. In contrast to the conventional template libraries of formal languages that require human effort in understanding the overall structure of the library to select and apply a proper template for a specific problem, the distinct characteristic of our approach is that all of the patterns are stored on computer as knowledge for creating effective guidances to facilitate the developers in writing formal specifications; they are “understood” only by the computer but transparent to the developers. Our pattern system is expected to support a systematic and automated formalization of informal requirements with the characteristic that the writer only needs to work on the informal level while an appropriate formal expression will be efficiently derived. Such a characteristic provides a possibility of automating the clarification process and motivates the adoption of the pattern system in supporting semi-automatic generation of formal specifications. This will allow the writer to concentrate only on the function design issues, while manipulation of formal notation to form the most appropriate formal expressions can leave to the machine. Consequently, the formalization process would become easier and mistakes would be reduced significantly.

Specifically, a *specification pattern* treats one kind of function as the composite of its attributes. It includes *derivation knowledge* for deriving the details of the function and *transformation knowledge* for representing the function properly in a formal notation. *Derivation knowledge* formally defines the necessary attributes to be clarified and a rule repository for each element which provides a set of clarification rules for guiding the assignment of the attribute. The formal semantics of each attribute ensures that the clarified requirements can be understood by machines and automatically transformed into formal specifications. *Transformation knowledge* is a set of transformation rules for determining the formal representation of the function according to the assigned attributes. To be "understood" by machines, the formal definition of pattern is given to guarantee the precision of the pattern system.

Based on the formal definition, the method for guiding requirements formalization by using the pattern system is described. It includes two steps: *requirements derivation* and *requirements translation.* The former guides the selection of appropriate specification patterns and applies the *derivation knowledge* of the selected patterns to guide the assignment of the defined attributes. The latter automatically transforms the assigned attributes into formal specifications according to the *transformation knowledge* of the selected patterns.

During the application process of the selected patterns, necessary data types will be automatically recognized and their definitions will be refined. Specifically, when applying each selected pattern, we use *function-related* *declaration* to guide the refinement of the related data types. It consists of two steps: *property-guided declaration* and *priority-guided declaration.* The former is applied during *requirements derivation* while the latter is carried out during *requirements translation.*

In addition to the design and definition of the pattern system, the representation of the pattern knowledge is also an important factor to the performance of the pattern-based approach. We adopt *attribute tree* and HFSM (Hierarchical Finite State Machine) to represent the pattern knowledge. *Attribute tree* organizes the attributes of a specification pattern into a tree structure to facilitate requirements clarification and understanding. A HFSM comprises a set of FSM (Finite State Machine) models to represent the knowledge to be used by machines in an individual specification pattern. These FSMs are organized in a hierarchy where the details of some portion of each high-level FSM are interpreted by a set of low-level FSMs. On the basis of the attribute tree and HFSM, several knowledge utilization algorithms are constructed. They retrieve pattern knowledge and accordingly produce appropriate guidance for formalizing requirements.

We also describe a prototype tool that supports the pattern-based approach. Specification pattern knowledge is stored in an XML file and the tool contains three major components manipulating on the file. *Knowledge extractor* retrieves appropriate knowledge from the XML file according to the value of a set of state variables. *Guidance generator* produces comprehensible guidance to the user on the basis of the retrieved knowledge. *Preprocessor* analyzes the user input responding to the produced guidance and transforms it into state variables' values that will be used by the *knowledge extractor.*

The major contributions of this thesis are summarized as follows.

- The design and formal definition of a *specification pattern system* for supporting requirements formalization

  A *specification pattern system* [37] for supporting requirements formalization is proposed. The novelty of the pattern system is that it is designed to be stored on computers as knowledge to produce guidance. Each specification pattern contains knowledge for guiding the clarification of necessary function details and the translation of the clarified details into formal specifications. The formal definition of the specification pattern is given to enable its automatic utilization. We also describe the structure of the pattern system where all the patterns are organized by classifying the functions they are used to formalize.

- A method for supporting requirements formalization by applying the *specification pattern system*

  Based on the *specification pattern system*, the method for refining informal requirements into formal specifications is described [38] [39]. It consists of two steps. The first step selects proper patterns from the pattern system and applies the *derivation knowledge* of each selected pattern to guide the developer to achieve necessary function details, which solves the first two problems in requirements formalization. The second step automatically transforms the obtained details into formal specifications using *translation knowledge*, which successfully tackles the third problem in requirements formalization.

- The design of a representation of the pattern knowledge

  To facilitate knowledge maintenance, utilization and understanding, we propose an approach to representing the pattern knowledge in *attribute tree* and HFSM. In this approach, attributes defined in each specification pattern are represented as an *attribute tree* that shows the definitions of these attributes in an intuitive way while preserving their formal semantics [40]. The knowledge for automatic utilization in each specification pattern is represented a HFSM where low-level FSMs interpret certain portion of the high-level FSMs [41]. Since many mature techniques have been developed for automatic manipulation on FSM, the pattern knowledge can be easily utilized, updated and verified.

- A tool that supports the pattern-based approach

  We have also implemented a prototype tool for supporting the pattern-based approach where the pattern knowledge is stored in a XML file [42]. By utilizing the knowledge, it derives informal requirements through interactions with its users on the semantic level and automatically transforms the obtained information into suggested formal specifications, which enables developers to concentrate on function issues without worrying about how to guarantee the completeness and how to formally represent these functions. The implementation of the tool demonstrates the validity of the approach and shows how the semiautomatic support can be provided for requirements formalization.

1.5 Summary

In this chapter, we first present the importance of requirements in software engineering and explain the merits of using formal specification technique to refine requirements. Then the problems to be handled in formalizing requirements into formal specifications are analyzed and summarized. Finally, the solution to these problems is briefly introduced and the major contributions of this thesis are presented.

In the next chapter, we will compare our research work with others from several aspects and show the novelty of the pattern-based approach.

Chapter 2

Related Work

Our pattern-based approach involves methods for supporting requirements formalization, requirements clarification, requirements formalization, data type declaration, knowledge representation and tool implementation for supporting requirements formalization. We discuss the related work in these relevant fields respectively.

2.1 Methods for supporting requirements formalization

Formal specification technique rises much attention in software engineering and many methods were proposed to support requirements formalization. These methods can be generally divided into two kinds. The first kind focuses on the construction of formal specifications and the second kind emphasizes the solution to the conversion of informal requirements to formal specifications. We discuss these two kinds respectively.

2.1.1 Methods for facilitating formal specification construction

There are two major kinds of methods for facilitating formal specification construction. One kind is to combine formal languages with informal notations or provide guidelines for writing qualified formal specifications. In [43], the authors describe some example approaches to integrating structured methods of software development with formal notations. Aiming at developing specifications that are both structured and formal, these approaches make the use of formal languages more acceptable to managers and engineers in software development organization. In [44], the authors use the structured analysis (SA) model of a system to guide the analyst's understanding of the system and the development of the VDM specifications. SOFL provides a SOFL formal specification language and a three-step approach to formal specification construction [45]. The SOFL language adopts graphical notation Data Flow Diagrams to describe system architectures, adopts Petri nets to provide an operational semantics for the data flow diagrams and adopts VDM-SL to precisely define the components in the diagrams. Such a language facilitates the structure design and overall understanding of the envisioned system, and guarantees the precision of the written requirements. The three-step approach treats formal specification construction as an evolutionary process where formal specification is built by gradually refining the ambiguities of the informal requirements. S. Liu [46] proposes an approach to constructing software specifications by integrating top-down and scenario-based methods. The top-down method is used to achieve a complete coverage of the user's functional requirements, while the scenario-based method is used to precisely define the functionality of each scenario and to construct complex scenarios by composition of simple scenarios. A software supporting tool for the method is also described.

Although formal language combined with informal notation is easier to use, there is no guidance for formal specification construction. On the other hand, the guidelines provided by the current methods are often given on an abstract level. Along the direction pointed out by these guidelines, developers still need to solve the bottom-level problems by themselves, which brings risk of erroneous formal specifications.

The other kind defines specification patterns and uses them to guide formal specification construction. This kind of methods has similar underlying theory as ours. Stepney *et al.* describe a pattern language for using notation Z in computer system engineering [47]. The patterns proposed are classified into six types, including presentation patterns, idiom patterns, structure patters, architecture patterns, domain patterns, development patterns. Each pattern provides a solution to a type of problem. Lars Grunske presented a specification pattern system of common probabilistic properties, called ProProST, for probabilistic verification [48]. They also give a structured English grammar that can guide in the specification of probabilistic properties. The pattern system and the structured English grammar capture expert knowledge and help practitioners to correctly apply formal verification techniques. However, a majority of these patterns cope with formal specification construction at a more abstract level compared with ours. They are not able to provide specific guidance for each step of the requirements formalization process.

There exist other example specification patterns that are intended to support the construction process at the bottom level are as follows. Ding *et al.* propose an approach for specification construction through property-preserving refinement patterns [49]. The refinement patterns are categorized into connector refinement, component refinement and high-level Petri nets refinement. Konrad *et al.* [50] analyze the timing-based requirements of several industrial embedded system applications and create real-time specification patterns in terms of three commonly used real-time temporal logics. They also offer a structured English grammar to facilitate the understanding of the meaning of a specification. This work is complementary to the notable Dwyer et al.'s patterns which are used for presentation, codification and reuse of property specification in a range of common formalisms [51]. In [52], the authors design a set of composable Timed Automata patterns based on hierarchical constructs in timed enriched process algebras. The patterns facilitate the description of complex systems using Timed Automata and can be used to transform CSP/TCOZ models to Timed Automata to enable the reasoning of TCOZ models. To solve bottom-level problems, these patterns are designed to deal with specific domains.

In spite of enthusiasm in academics, specification patterns are not yet widely utilized in industry mainly because of the difficulties in applying them. Effective applications of most specification patterns require full understandings of them, and the ability to select and solve their specific problems depends on the understanding, since their informal representations make it impossible to utilize the pattern knowledge without human involvement. Developers need to study the pattern knowledge and gain enough experience in applying the provided general solution to specific problems. Statistical data shows that large amount of patterns have been developed, but only a small subset of these patterns are being used by industry practitioners and most of them are wasted as users may not fully understand how to leverage them in practice [53]. In comparison with the above related work, the novelty of our approach is that it treats the formally defined patterns as knowledge that can be automatically analyzed and utilized by machines to generate comprehensible guidance and the patterns remain transparent to the users. The produced guidance help the users, in an interactive manner, clarify ambiguities and gradually refine informal requirements into formal specifications. Thus, developers need neither to be educated on the patterns nor to be trapped in tedious and sophisticated formal notations; they only need to make critical decisions on the semantic level and are able to focus on function design issues. With the help of the guidance produced by our approach, developers who are not familiar with formal notations are also able to construct reasonable formal specifications.

There is a specification pattern similar to ours proposed in [54]. It is also defined with formal semantics for automatic utilization. However, their patterns are designed only for a specific domain: formal specification of OCL constraints. They can only be described in the context of UML and can only be used by UML experts. By contrast, our patterns are aimed at dealing with commonly used functions and allow new ones to be designed to handle wider range of functions. Users can focus on function design of the envisioned system and leave the syntax issues to the tool. Moreover, the user of the OCL specification patterns is asked to provide all parameters of the intended requirement at one time and an OCL constraint will be automatically generated. This is good enough for OCL constraints but may have problems for complex functions. The reason is that in most cases, it is very difficult to determine all the parameters of a requirement at one time, especially for complex functions. We believe that requirements formalization is also a clarification process. Thus, our patterns are designed to guide the clarification of the intended requirements step by step and collect the provided function details for further formalization.

2.1.2 Methods for transforming informal requirements into formal specifications

Compared with formal notations, natural language is much easier to comprehend and use. Hence, many researchers are devoted in automatic or semi-automatic transformation from informal descriptions to formal specifications [55] [56], with which designers or developers are allowed to remain documenting in natural language, as they wish. And the documented informal description can be formalized automatically. William E *et al.* introduce a general framework for formalizing a subset of UML diagrams in terms of different formal languages based on a homomorphic mapping between metamodels describing UML and the formal language [57]. Cory Plock *et al.* show how to transform LSC (Live Sequence Charts) specifications with concurrency to timed automata [58]. Sunil Vadera *et al.* propose an interactive approach for producing formal specifications from English specifications [36]. Several tools have also been proposed for automatic transformation, such as U2B [59] and RoZ [60].

Some of these transformation methods only translate requirements from one language to another based on certain pre-defined syntactic rule without considering the real meaning of the models. They are not able to analyze the semantics of the involved functions. Other methods usually involve NLP (Natural Language Processing) and their performance largely depends on the effectiveness of such technique. However, the current NLP is not satisfactory for automatic analysis of informal specifications and still facing the problem of the great amount of ambiguity issues at every linguistic level of natural language. The correctness of the transformation result is hard to guarantee. Although controlled languages are proposed to reduce these ambiguities to some extent [61] [62], such as AECMA Simplified English, by restricting the grammar and vocabulary, it introduces other shortcomings. First, designers have to learn the standard of the controlled language so that they will not write any expression determined as illegal input that can not be recognized by machines. Secondly, the preciseness of statements will be affected if the words or expressions that the designer prefers are not included in the controlled language. Lastly, language completeness is hard to evaluate and guarantee since it is almost impossible to summarize all the combinations of words in a language and extract a subset of them for describing all possible specification functions. In [62], the authors give an example that shows even the AECMA Simplified English would make certain informal idea unable to be expressed. There are also some researchers proposing intermediate languages to bridge the gap between informal and formal descriptions [63] [64] [65]. However, these languages still require developers to deal with many details in formal notations when reaching the bottom level of the intended formal specifications. Therefore, there is no effective tool-support in constructing formal specifications on the semantic level, but the introduction of patterns seems to offer a solution. By contrast, informal descriptions is not treated as the resource of our pattern-based approach. The desired functions are obtained by gradually clarifying informal ideas with human involvement. Thus, the performance of the approach will not be affected by NLP technology.

2.2 Tool support for requirements formalization

Several kinds of tools have been developed for supporting requirements formalization.

- The first kind supports the writing of formal specifications in different languages. For example, Z User Studio is an integrated Z support tool [66]. It supports the production of well-formed Z specifications by providing facilitates for building, editing, checking and reviewing Z specification documents. The Rodin Platform is developed for supporting refinement and mathematical proof on Event-B [67]. The refinement function represents the envisioned system at different abstraction levels and the mathematical proof function verifies consistency between refinement levels. VDMTools supports software development based on the specification written in VDM-SL or VDM++ [68]. It contains a syntax checker and type checker for improving the quality of the written VDM specification. These tools provide effective support for editing formal specifications without syntax problems, but lack of intelligent support for guiding the requirements clarification and automatic translation of requirements in formal notations.

- The second kind aims at automatic transformation from informal requirements to formal specifications. U2B translator [59] converts UML Class diagrams, including attached state charts, into the B notation [69].

RoZ produces a formal specification from an annotated specification by translating the UML constructs and merging them with the annotations [60]. They allow developers to model the target system with their preferred language, which is much easier to use and comprehend, and provides a automatic mean to transform the model into formal specifications. However, they are not working on the semantic level and can hardly give any guidelines on how to clarify requirements for their formalization. Developers who want to use these tools need to explore the necessary function details according to their own experience. If the manually explored function details are inadequate for formalization or involve ambiguities, these tools will fail to produce the target formal specifications. By contrast, our tool that implements the pattern-based approach is able to support the overall requirements formalization process from requirements clarification to formal specification generation.

- The third kind supports the formalization of system properties. SPIDER [70] derives and instantiates system properties in terms of their natural language representations. Prospec [71] is developed to assist developers in the elicitation and specification of system properties based on Specification Pattern System and Composite Propositions. Our tool differentiates from them by the ability to support the formalization of requirements involving complex data types.

- The fourth kind deals with formal specifications of data types. Kanth Miriyala *et al.* describe an interactive system called SPECIFIER for deriving formal specifications of data types and programs from their informal descriptions [55]. Compared with this tool, our tool supports requirements formalization at a higher level of abstraction so that the developers can focus on the semantics of real functions without the need of considering their data type issues.

2.3 Methods for supporting requirements clarification

As a critical step in requirements formalization, requirements clarification is usually separated from requirements formalization. Gerald Kotonya *et al.* present how and which methods should be used for each activity of requirements engineering including requirements clarification [72]. Eric Knauss *et al.* propose an approach to analyzing online requirements communication and a method for the detection and classification of clarification events in requirement discussions [73]. Jawed Siddiqi *et al.* describes their work towards a system that judiciously combines the strengths of formal specification and prototyping to assist in the construction, negotiation, clarification, discovery and formalization of requirements [74]. However, few of them aim at clarifying requirements for automatic transformation into formal specifications .

On the other hand, several literatures, as mentioned in the previous section, claimed the proposals of effective automatic formalization methods for requirements clarified in different informal notations. Requirements clarification using the above informal notations needs to be manually conducted and the quality of the clarified requirements are hard to guarantee. Besides, most of the informal notations aim at modeling envisioned system on a relatively abstract level and the clarified requirements are often lack of necessary details to be automatically formalized. Although natural language is capable of describing requirements in detail, *NLP* technique for natural language understanding and analysis is still not satisfactory [61], especially when the involved data structures and function details are complicated.

There are also some researches on the design of new languages that facilitate both requirements clarification and automatic formalization. Lingzi Jin *et al.* describes the NDRDL language and the system NDRASS for automatic generation of formal specifications in Z from requirements definitions in NDRDL [75]. In [76], the author proposes a new requirements language to better structure informal requirements, and shows the method for transforming requirements written in this language into formal specifications. These new languages contribute to the intuitiveness of the requirement clarification process and step further to formal specifications compared with other existing informal representations. But they are designed to model system architectures without considering the included function details and rich data types. By contrast, our approach is able to deal with the clarification of detailed behaviors for data-intensive systems.

2.4 Methods for data type declaration

We know of no existing approach that provides assistance throughout the whole data type declaration process, although some researches have been concerned with certain aspects of the problem.

Type checking technique and model transformation have been introduced to facilitate data type declaration [77] [78] [36]. The former detects static type errors to prevent erroneous formal descriptions while the latter allows data to be described in certain intermediate language easier to use and provides a method for transforming the data model into formal data types. Unfortunately, they fall short of meeting practitioners' demand. First, relations between types and functions to be described is not considered, i.e., type definitions incapable of or unsuitable for describing the intended functions are not able to be identified. Secondly, no guidance or automated assistant is provided during the declaration process. Lastly, the consistency between formal expressions and type definitions cannot be guaranteed. In a formal specification f, if a type definition t is changed into t', all the formal expressions involving state variables defined with t need to be manually modified to be consistent with the new definition t'.

In the type checking field, several typecheckers are designed and implemented for various formal specification languages with different type systems. Jian Chen et al. [78] develop a simple but useful set of rules for type checking the object-oriented formal specification language Object-Z and an earlier version of the type checker for Z is given in [79]. For the Vienna Development Method (VDM), the most feature-rich analytic tool available is VDMTools which includes syntax- and type-checking facilities [77] [80] where syntax checking results in positional error reporting supported by an indication of error points and type-checking can be divided into static type-checking and dynamic type-checking. The former checks for static semantics errors of specifications including incorrect values applied to function calls, badly typed assignments, use of undefined variables and module imports/exports, while the latter aims at avoiding semantic inconsistency and potential sources of run-time errors. As one of the major components in the Rodin tool for Event-B, static checker analyses Event-B contexts and Event-B machines and generates feedback to the user about syntactical and typing errors in them [81] [67]. Prototype Verification System (PVS) extends higher order logic with dependent types and structural and predicate subtypes. In addition to conventional type-checking, it returns a set of proof obligations TCCs (Type Correctness Conditions) as potent detectors of erroneous specifications and provides a powerful interactive theorem prover that implements several decision procedures and proof automation tools [82] [83]. In [84], the authors present a type checker for formal specifications of software systems described in Real-Time Process Algebra, which is able to handle three tasks: identifier type compliancy, expression type compliancy and process constraint consistency. In [85], the authors define the type system of formal language Circus which combines Z, CSP and additional constructors of Morgan's refinement calculus, and describes the design and implementation of a corresponding type checker based on the typing rules that formalize the type system of Circus.

The quality of the declared data types can be significantly improved by the supporting tools listed above, unfortunately practitioners are still complaining about the difficulties in identifying real objects by formal data type definitions due to the lack of effective guidelines throughout the declaration process and everlasting appearance of errors implicitly explained. Despite the use of "semantic analysis" in some of these tools' underlying theories, it refers to the semantics of the embedded type system that is part of the built-in mechanism, rather than the semantics of the informal requirements in users' mind. By contrast, our approach tries to connect the semantics of specifications with the corresponding system behaviors through data types and evaluate the appropriateness of the declared types on the real semantic level. Moreover, the given systematic guidance in the overall declaration process specifies how to reach the appropriate data types step by step while checking the correctness of the result of each step, which alleviates burdens of manual design.

There are also some researches done for transforming models in intermediate languages to formal data type definitions. These intermediate languages provide accessible visualization of object relation models and therefore simplify the object identification process. In [36], entity relationship models are treated as the basis for producing VDM data types in specifications. Colin Snook et al. [86] propose a formal modeling technique that emerge UML and B to benefit from both languages where the semantics of UML entities is defined via a translation into B. [87] presents an automated transformation method from UML class diagrams with OCL constraints to Alloy which is a formal language supported by a tool for automated specification analysis. The problem, however, lies in the fact that identifying and defining objects are separated from the functions to be described in these methods and totally depend on the developer's initial understanding of the real system. Hence our approach would be more reliable in declaring data types for function description and practitioners can utilize models in graphical representations as supplementary materials.

2.5 Methods for pattern knowledge representation

Many efforts have been made in the field of specification knowledge in terms of patterns. As previously mentioned, most kinds of pattern knowledge are represented informally in order to facilitate understanding since they are generally defined with common solutions to common problems and their application to specific problems depends on how well the users understand the pattern knowledge. They are unsuitable for automatically processing by machines. By contrast, our specification pattern knowledge is formally represented and can be recognized by machines without ambiguity, which provides a possibility to automate the requirements formalization process. Developers who intend to benefit from the knowledge are only required to follow the produced guidance without the need of knowledge learning.

Meanwhile, researchers are still improving the usability of such patterns by specifying pattern solutions with pattern specification languages [88] [89] [90] [91] and some of them provide formal semantics for existing patterns. *Elemental Design Patterns* are proposed in [92] to provide a formal semantics for composition of OO software architecture. It bridges the gap between the abstraction of design patterns and the reality of working with an ultimately mathematically expressible system such as code. In [93], the authors introduce a set of commonly used process change patterns for PAIS (process-aware information system) engineers to facilitate the comparison between different approaches for process modeling. They provide the formal semantics of these patterns to ground pattern implementation and pattern-based analysis of PAISs on a solid basis. The authors in [94] try to define the assembly patterns with process algebra theory, so that the component assembly pattern in the software architectures can be described more precisely and better analyzed. The difference between these researches and ours is that they provide formal semantics for patterns to pull the provided solutions closer to the specific problems in real settings while our formal representation is to facilitate the maintenance and utilization the pattern knowledge on machines and the understanding of the produced guidance for users.

There are also other kinds of formal representations for requirements formalization knowledge, such as the knowledge in the previously mentioned work [36], [57], and tools U2B and RoZ. But this kind of representation only needs to describe a set of pre-defined syntactic mapping rules without considering the real meaning of the relevant requirement details. Our knowledge is much more complex and needs a representation that can organize all aspects of the knowledge in a clear and accurate manner. The attribute tree provides a clear view on requirements structure and a formal semantics of the involved attributes. The relations between the FSMs in each HFSM represent the architecture of the pattern knowledge and the symbols involved in each FSM represent the elements for composing the knowledge.

2.6 Summary

In this chapter, our research is divided into several aspects. For each aspect, we discussed the related work and explicitly explained the differences between our approach and these related work.

From the next chapter, we will begin to present the pattern-based approach. In the next chapter, the underlying principle and outline of the approach is given based on some preliminaries. Then an example system is introduced for later explanation of the approach details.

Chapter 3

An overview on the pattern-based approach to requirements formalization

This chapter gives an overview on the proposed approach. Before presenting the outline of the pattern-based approach, some basic concepts need to be first introduced.

3.1 Preliminaries

Two concepts are explained in this section. One is SOFL (Structured Object-Oriented Formal Language) including SOFL language and SOFL three-step approach. The underlying theory of our approach is language-independent, but a specific formal notation is necessary for illustrating how the approach works. We choose SOFL as the example notation due to our expertise. The other concept is *pattern*. We briefly introduce the history of pattern and present how we are inspired by this concept.

3.1.1 SOFL

SOFL is composed of two parts: a formal but comprehensible language for requirements and design specifications, called SOFL specification language and a practical method for developing software systems, called SOFL method. We will explain these two concepts respectively.

The SOFL language is an integration of three notations, including Data Flow Diagrams, Petri nets, and VDM-SL. Specifications in SOFL are usually composed of modules that are associated with CDFDs (Condition Data Flow Diagrams). CDFDs are designed in a hierarchy to describe the architecture of the system under design, while the components used in each CDFD, such as data flows, data stores, and processes, are precisely defined in the

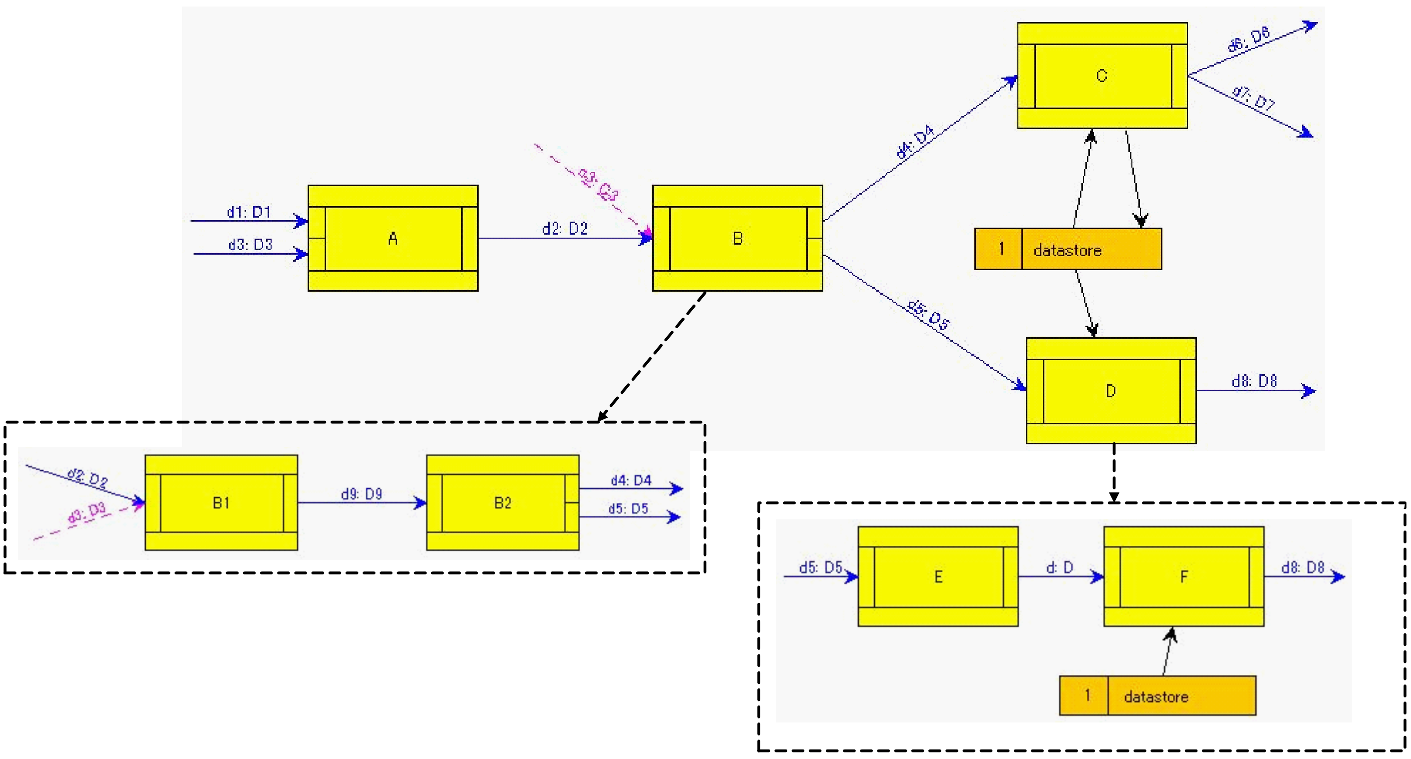


Figure 3.3. An example CDFD hierarchy

corresponding module.

Figure 3.3 shows an example CDFD hierarchy where each box in light yellow denotes a process and each box in orange denotes a data store. Processes are connected by directed lines where each line denotes a data flow and the attached label denotes the name and data type of the data flow.

This hierarchy includes two levels where the inner structures of the processes B and D are described by two lower-level CDFDs. In the top level CDFD, four processes A, B, C, D, and one data store *datastore* are involved. The process A has two input ports receiving data flows $d1$ and $d3$ respectively, which means that only one of the two data flows can be consumed as input for producing output. When either $d1$ or $d3$ is available, process A is activated and takes the available data flow as input and produces $d2$ as output. Process B will be activated when receiving the produced $d2$ and control flow $c3$. It owns two output ports producing data flows $d4$ and $d5$ respectively, which means that only one of the two data flows can be produced as output. If $d4$ is produced, the process C will be activated and output data flows $d6$ and $d7$. If $d5$ is produced, the process D will be activated and output data flow $d8$. Note that both the processes C and D are connected to *datastore*. The two directed lines between C and *datastore* indicate that C uses the data of *datastore* and also updates *datastore* during its execution. For the process D, there is only one line originated from *datastore*, meaning that D only uses the data of *datastore* during its execution.

The process B is decomposed into a CDFD composed of processes $B1$ and $B2$. This CDFD illustrates how the input $c3$ and $d2$ are transformed into $d4$ or $d5$ within the process B. These two data flows are first transformed into $d9$ by process $B1$ and then turned into $d4$ or $d5$ by process $B2$. Similarly, the process D is decomposed into the processes E and F where E transforms input $d5$ into d and F transforms d into output $d8$. The process F reads from *datastore* during its execution.

A module, associated with a CDFD, denotes a set of inter-related system behaviors that are relatively independent from others. It encapsulates data and processes used in the associated CDFD in a pre-defined structure. All of the data flows and store variables are declared using well-defined types. Each process denotes an operation that absorbs input and produces output, which is specified in terms of pre- and post-conditions. Input must satisfy the pre-conditions while output must satisfy the post-conditions. Figure 3.4 shows the modules for defining the example CDFD in Figure 3.3 where the module $TopCDFD$ defines the top level CDFD, the module B_decom defines the CDFD of the process B and the module D_decom defines the CDFD of the process D.

Each module mainly comprises five portions: *const* portion for constant declaration, *type* portion for type declaration, *var* portion for variable declaration, *inv* portion for defining invariants and the portion for defining processes. We take the module *TopCDFD* as an example. Its involved types are declared in type portion such as *D1* and *D2*. Datastore *datastore* in the top level CDFD is defined in the *var* portion as a variable. Each process is defined with input, output, pre- and post-conditions. The statement "$decom : B\_decom$" in the process B and statement "$decom : D\_decom$" in the process D indicate that the processes B and D are decomposed into the modules B_decom and D_decom respectively. One can refer to the bodies of these two modules for the detailed description of the process B and D.

Since data type plays an important role in our approach, the type system in SOFL is briefly introduced. In the module structure, the *type* portion is composed of a set of user-defined types for declaring the variables and data flows in the module. These types are defined based on the built-in types in SOFL. There are two kinds of built-in types. One is basic types including numeric types, boolean type, character type and enumeration types. There are

```
module TopCDFD;
const;
type
D1 = …
D2 = …
… ;
var
datastore: … ;
inv;
process A(d1: D1 | d3: D3) d2: D2
pre
post
end_process;
process B(d2: D2, c3: C3) d4: D4 | d5: D5
decom: B_decom
end_process;
process C(d4: D4) d6: D6, d7: D7
ext wr datastore
pre
post
end_process;
process D(d5: D5) d8: D8
decom: D_decom
end_process;
end_module;
```

```
module B_decom;
const; type; var; inv;
process B1(d2: D2, c3: C3) d9: D9
pre
post
end_process;
process B2(d9: D9) d4: D4 | d5: D5
pre
post
end_process;
end_module;
```

```
module D_decom;
const; type; var; inv;
process E(d5: D5) d: D
pre
post
end_process;
process F(d: D) d8: D8
ext rd datastore
pre
post
end_process;
end_module;
```

Figure 3.4. The module specification of the example CDFD hierarchy

four numeric types: *nat*0 denoting natural numbers including zero, *nat* denoting natural languages, *int* denoting integers and *real* denoting real numbers. The other kind is compound types including set types, sequence types, composite types, product types, map types and union types. Table 3.1 shows the constructor of each compound type.

The SOFL method is a three-step modeling approach, transformation from structured design specifications to object-oriented implementations, and specification-based inspection and testing for program verification and validation. In this approach, the modeling of a system is an evolutionary process, starting from building an informal specification, through transforming it to a semi-formal one, and finally constructing a formal specification. In the informal stage, requirements are written in natural language that reflects the desired functions of the system, data resources, and necessary constraints. Based on the informal specification, the system can be designed into modules in semi-formal specification in which all data types are defined formally; all variables are declared using well-defined types; and all processes are specified in terms of pre- and post-conditions but these conditions are usually written informally (which reflects the semi-formal feature). The formal specification is then derived from the semi-formal one by formalizing its informal parts.

Table 3.1. The constructors of the compound types in SOFL

Type	**constructor**	**Example**
set	set of	$t = set\ of\ T$ (Type t is a set type where each element is of type T)
sequence	seq of	$t = seq\ of\ T$ (Type t is a sequence type where each element is of type T)
composite	composed of f1: T1 ... fn: Tn end	$t = composed\ of$ $f1 : T1$ $f2 : T2$ $f3 : T3$ end (Type t is defined as a composite type with three fields $f1$, $f2$ and $f3$. These three fields are defined as types $T1$, $T2$ and $T3$ respectively. Any value of t type is a composite object composed of three attributes $f1$, $f2$ and $f3$. Each attribute fi of the value is specified with a value of Ti type.)
product	T1*...*Tn	$t = T1 * T2$ (Any value of t type is a tuple $(v1, v2)$ where each vi is of Ti type)
map	map ... to ...	$t = map\ T1\ to\ T2$ (Type t is a maximum mapping from $T1$ to $T2$)
union	T1 | ... | Tn	$t = T1 \mid T2 \mid T3$ (Any value of t type can come from one types $T1$, $T2$, $T3$)

Our research also adopts this idea that requirements formalization should start from a general intention and the ambiguities of the original requirements be gradually resolved as the developer gradually clarifies his intended requirements. The *specification pattern system* categorizes all the patterns according to the functions they are used to formalize and pattern selection is actually a process of clarifying the general intention of the developer. The application of the selected pattern guides the assignment of the relevant attributes sequentially, which is actually to guide the clarification of the ambiguities involved in the original requirement step by step.

The reader who wishes to understand more details of SOFL can refer to [45] for extensive reading.

3.1.2 Pattern

The concept of pattern was initially introduced by Alexander et al [95] in which patterns are used to share the author's intelligence on handling the large-scale structure of the environment. A pattern is created to convey feasible solutions to the corresponding re-occurred problem within a particular context, which is intended to facilitate people who face the same problem. Instead of learning issues like the growth of town and country, the layout of roads and paths, etc, software industry was inspired to make their own patterns for software design, which has proved to be useful [53]. One of the well-known achievements is *design patterns*, focusing on software design problems [96]. Several books [96] [97] about UML based object-oriented design with patterns are published, aiming at promoting design patterns among practitioners to help create well crafted, robust and maintainable systems. Amplifying the benefits of patterns, researchers and practitioners have later made it applicable in many other areas of software development process, including formal specification construction.

Inspired by the underlying theory of design pattern that classifies system architecture into different kinds and provides a general design plan for each of these kinds, we start to analyze the feasibility of classifying commonly used functions and provides a formalization framework for each kind of function.

A pattern usually consists of three parts.

- intention – describing the problem or a sort of problems that the pattern intends to deal with
- solution – presenting recommended algorithms or methods for solving the problem
- context – illustrating the condition under which the pattern can be utilized.

Pattern structure, however, varies depending on the specific situation it applies to. In our research, patterns are designed to be applied by a potential tool to generate appropriate guidelines for the developers. For this reason, the structure of our pattern must be designed to be easily processed by software tool. Besides, all of the individual patterns must be organized properly so that they can be easily identified, applied, updated, and extended. Solutions to these problems are based on the classification of patterns and the structure of each individual pattern that are presented in the next chapter.

3.2 The outline of the approach

The outline of our pattern-based approach is given in Figure 5.15 where requirements formalization consists of two stages: *requirements derivation* and *requirements translation.*

During the first stage, the informal requirements in the developer' mind is gradually clarified and the function details of the clarified requirements is derived. There are two major activities in this stage: *attributes clarification* and *data type declaration.* In the first activity, each function of the requirements is regarded as the composition of a set of attributes and clarifying a function is to specify each attribute for composing the function. In the second

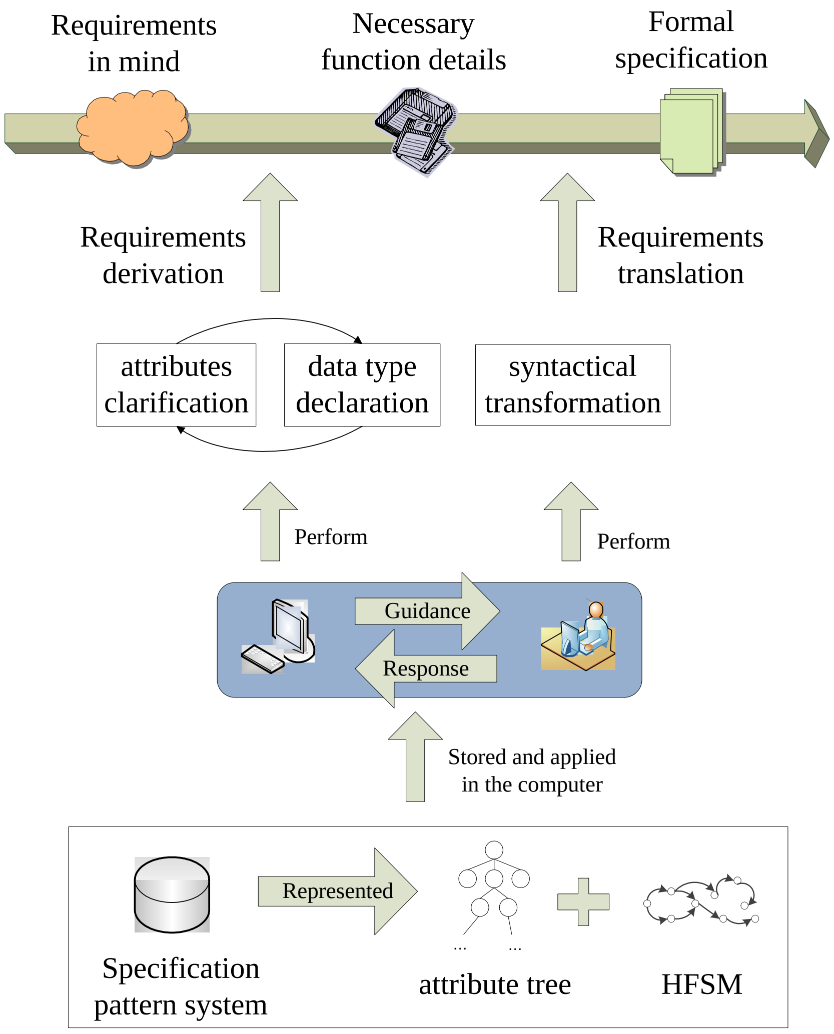


Figure 3.5. The outline of the pattern-based approach

activity, all the necessary data types for formalizing the requirements are declared [98]. Instead of being performed independently, these two activities are interleavingly carried out. Clarifying attributes needs to use the existing types defined in *data type declaration*. Meanwhile *attributes clarification* help specifies the types that should be defined to enable the description of the clarified attributes. Therefore, the two activities precede hand in hand until both them are terminated. In the second stage, the obtained function details are translated into a formal specification. Such a translation is actually done by a syntactical transformation from the format of the function details to a formal notation.

Both of the stages are performed through interactions between the developer and computer where computer produces guidance and the developer inputs the response to the computer. The response triggers the computer to produce new guidance which is then followed by the developer for the next step of requirements formalization. Such a process repeats until a formal specification is achieved. The foundation of this interaction process is a *specification pattern system* which is stored on the computer to be applied to interact with the developer. The *specification pattern system* organizes a set of *specification patterns* in a hierarchical structure where each pattern carries two kinds of knowledge for formalizing one kind of function: *derivation knowledge* and *transformation knowledge*. The former is designed to support requirements derivation and the latter is created to deal with requirements translation. We adopt *attribute tree* and HFSM to represent the above pattern knowledge in computer. *Attribute tree* is used in *derivation knowledge* to intuitively show the definitions of the requirement attributes that need to be clarified, which facilitates the developer's understanding on the structure of the intended requirements. HFSM is used to describe the knowledge for guiding the clarification of the attributes and generating the target formal specification, since these knowledge is only applied by machines and HFSM is easy to be manipulated automatically and maintained by several mature supporting tools.

It should be noted that the requirements formalization process in our approach is not expected to be fully automatic due to the need for human decisions, but it is expected to help the developer clarify ambiguities in the informal requirements and generate appropriate formal specifications.

From the next chapter, we will explicitly describe each component in the approach outline from the bottom level since the understanding of the high-level components relies on that of the lower-level components. Consequently, the structure and definition of the *specification pattern system* will be first introduced. Then we will show how to apply the pattern system for supporting the two stages during requirements formalization. The first stage requirements derivation is described by presenting the inter-related activities attributes clarification and data type declaration, while the second stage requirements translation is described by explaining the syntactical rules for transforming the obtained function details into formal specifications. The representation of the included pattern knowledge is finally described since it is designed based on the application process of the pattern system.

3.3 An example system

To illustrate our approach more clearly, a banking system is introduced and its architecture is shown in Figure 3.6. When presenting the details of each technique in the approach, we will choose some of the included functions as examples to demonstrate the application of the technique in real settings.

The banking system manages a set of bank accounts in the datastore *account_store* where each account is owned by one customer with a unique pair of account number and password. An account holds the information of balance and a sequence of transactions. Balance reveals the current amount of each kind of currency and each transaction records the date, type, currency type and the amount of certain operation performed by the customer. The date of today is hold by the datastore *today* and the rate information for currency exchange is carried by the datastore *rate_store*.

According to the CDFD, the system provides services for two roles: one is the bank customers and the other is the manager of the banking system. Process *roleSelection* receives the role information and accordingly determines the next operation to be performed. For each customer, the banking system will first perform identity validation (denoted as process *Account_ confirm*) to check whether the current customer is eligible to receive the services. An authorized customer can choose from four services: *deposit*, *withdraw*, *information display* and *currency exchange* (denoted as processes *deposit*, *withdraw*, *display* and *exchange* respectively). The service *deposit* allows the customer to deposit money in their own accounts. By the service *withdraw*, the customer can withdraw from their own accounts. The service *information display* shows the required information to the customer, such as balance and transaction history. The service *currency exchange* allows the customer to exchange between several kinds of currencies.

For the manager with valid ID and password, the banking system provides four services including *balance*

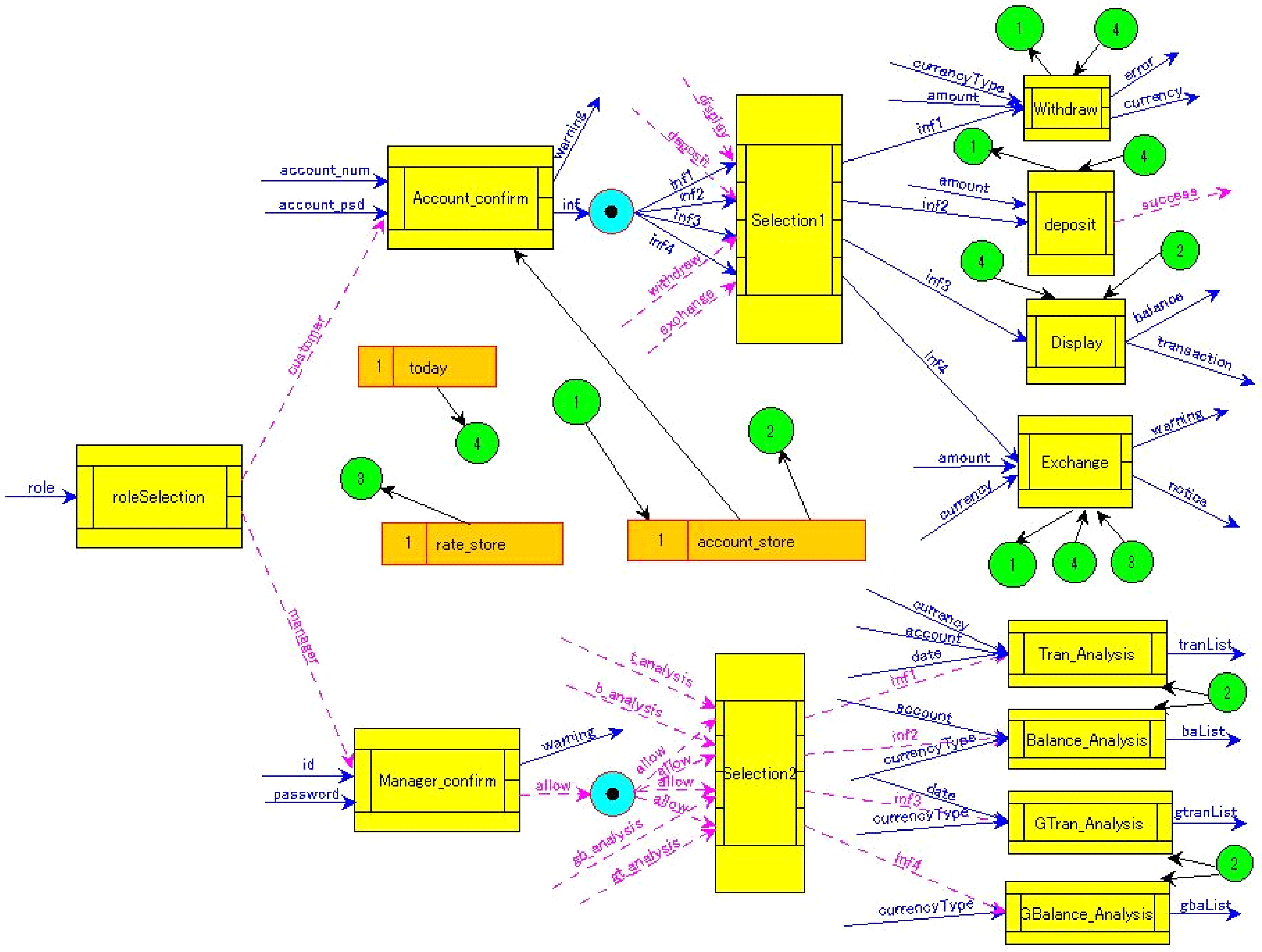


Figure 3.6. The CDFD of the banking system

analysis, *transaction analysis*, *global balance analysis* and *global transaction analysis* (denoted as processes *Balance_Analysis*, *Tran_Analysis*, *GBalance_Analysis*, *GBalance_Analysis* respectively). The function *balance analysis* displays the balance of the designated currencies in a designated account; the function *transaction analysis* lists the desired transactions in a designated manner; the function *global balance analysis* shows the sorted balance information of all the accounts; and the function *global transaction analysis* exposes the sorted transaction information of all the accounts.

3.4 Summary

In this chapter, we first introduce the basic concepts used in our requirements formalization approach, including the formal notation SOFL language that is adopted for illustrating our approach in the thesis, the SOFL three-step approach that inspired us in the development of our approach and pattern that acts as the foundation of our approach. Then we describe the outline of our requirements formalization approach which provides a clear view on the overall framework of the approach. This outline will help readers better understand each involved component. Finally, a banking system is introduced and its architecture is shown in a CDFD diagram.

From the next chapter, we will begin to explicitly describe the components involved in the approach outline. Serving as the foundation of the approach, *specification pattern system* is first introduced.

Chapter 4

Specification pattern system

Specification pattern system is composed of a set of *specification patterns* each dealing with the formalization of one kind of function. Instead of isolating from each other, these patterns are connected in a way that one pattern adopts other patterns to formalize certain sub-functions. All the patterns are categorized according to the kinds of function they can be used to formalize. We will first present the concept of *specification pattern* and then show how these patterns are organized in the hierarchy of the pattern system.

4.1 Specification pattern

The idea of adopting pattern in requirements formalization originates from the fact that most requirements can be divided into a set of bottom-level functions and the bottom-level functions of the same kind share the same set of attributes. If the attribute set for each kind of bottom-level function is obtained as a pattern of the function description, the clarification of requirements can be guided, as well as requirements formalization. This underlying theory of our *specification pattern* is shown in Figure 4.7.

A requirement is the combination of functions and a function is composed of its attributes. Individual functions of the same kind are composed of the same set of attributes, which forms patterns as guidance for clarifying the individual functions of this kind. For each kind of function fi, each its attribute $attr_{ij}$ can be defined as a union type consisting of a set of constituent types $\{t_{ij1}, t_{ij2}, ...\}$. These constituent types indicate different ways for specifying $attr_{ij}$, contributing to individual functions with different attribute values. Thus, given two requirements r and r' which are both composed of functions f and f', if f or/and f' in r has at least one attribute specified differently from that of f or/and f' in r' , r and r' are two different requirements. The same example can be found in Figure 4.7 where requirements $r1$ and $r2$ are built by different combinations of $f1$ and $f2$ because of their

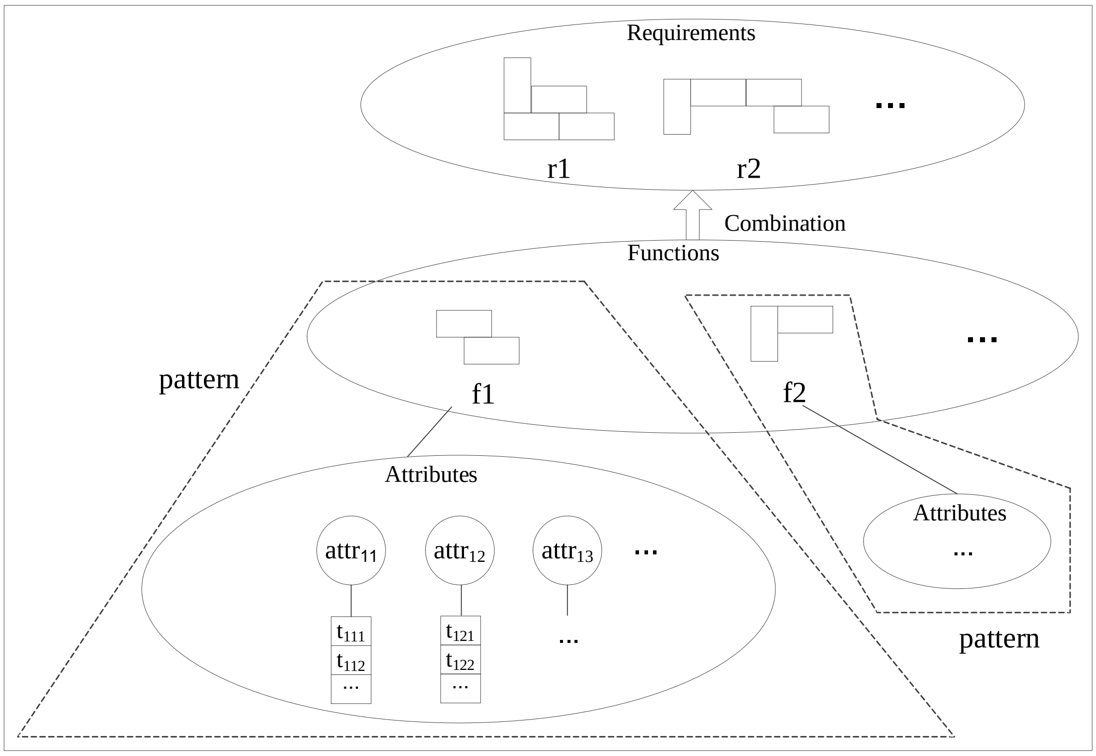


Figure 4.7. The underlying theory of the pattern-based approach

differently specified attributes.

Consider the function of altering data items of a system variable. It includes two attributes: the system variable to be altered (denoted as $objAttr$) and the way to alter it (denoted as $howAttr$). All the specific functions that describe the altering of data items of a system variable are composed of these two attributes. Therefore, a pattern can be created which defines attributes $objAttr$ and $howAttr$ as the necessary function details that must be clarified to obtain a complete description of alter functions. When applying the pattern to formalize a specific alter function, one should clarify these two attributes according to their definitions. Let's take the attribute $howAttr$ as an example. It is differently specified in different individual altering functions. If the individual function replaces the target system variables with new values, the attribute should be specified as these new values. However, if the intended individual function only modifies parts of the target system variables, the attribute needs to be specified in several aspects, such as the identification of the data items to be modified in the target system variables and the way to alter the identified data items.

Based on the underlying theory, we start the introduction of our pattern from its structure for better understanding the inner mechanism. Then the definition of the pattern is presented.

4.1.1 Pattern structure

Similar to traditional patterns, our *specification pattern* is also designed with a structure for organizing the included knowledge. It is composed of the following four items:

name	the unique identity of the pattern
explanation	explains what kind of function the pattern can be used to formalize
constituents	specifies how to write the requirement for the intended function
solution	rules for transforming the achieved requirement into formal expressions

As can be seen from the structure, a *specification pattern* is established to guide the formalization of one kind of function f. It mainly consists of two parts providing solutions for tackling the two tasks during formalization: the clarification of f and the representation of the clarified f in a formal notation. In the first part, f is treated as a composition of its necessary attributes. These attributes are formally defined as elements and clarifying f is to assign values to the elements according to their definitions. A set of clarification rules are provided for guiding such assignments. In the second part, a set of transformation rules are given for generating formal representation of f according to the values assigned to the elements. Consider the function "belong to" which describes a relation where certain object is a member of another object. Each specific "belong to" function is composed of two attributes: the member and the object that the member belongs to. Therefore, the corresponding pattern includes clarification rules for guiding the assignment of the two attributes and transformation rules for generating a formal representation of the "belong to" function according to the assigned values.

Figure 4.8 shows an example pattern. The *name* of the pattern is "sorting" and the *explanation* item tells that it is used to describe the function of placing objects in a particular order.

The *constituents* item specifies how to derive the requirement for a sorting operation by providing *derivation knowledge*: item *elements* composing a set of elements where each element denotes one of the attributes of sorting functions and item *rule for guidance* comprising a set of clarification rules for assigning values to these elements. There are four elements in the item *elements*: *objs* denoting the set of objects to be sorted, *result* denoting the variable representing the result of the sorting operation, *ruleType* denoting the category of the sorting rule, *rule*

```
name          sorting
explanation   Placing objects in particular order
constituents
    elements:
      objs, result, ruleType, rule
    rule for guidance:
      1. if the data type of the given value of objs is char
           then ask the developer to assign another value to objs

      2. if the given value of objs owns two sub-objects, element
           ruleType should be assigned as “ et”

      ……
solution
    1.  if    the pattern is used to describe certain sub-function for
              another pattern
          and   the value assigned to element result is a defined variable
          and   the value assigned to element ruleType is “ etg”
        then    the resultant formal expression is :
                  the formal expression generated by applying pattern
                  group with its element objs assigned as objs in this
                  pattern and result assigned as “ elems(result))” + “  and rule”
     ……
```

Figure 4.8. Pattern ”sorting”

denoting the rule for sorting *objs*. In *rule for guidance*, two example rules are listed to illustrate the meaning of the item. If the specified *objs* is a character, the first rule will be applied and the developer will be asked to re-specify *objs* since sorting one character makes no sense. Otherwise, if the specified *objs* is a set owning two members, the second rule will be applied and element *ruleType* will be automatically assigned as "*et*", meaning each pair of neighbor objects in *result* holds the same relation.

Item *solution* involves a set of transformation rules for determining the resultant formal expressions based on the values assigned to the four elements. For example, the first rule gives the formal expression for the elements whose values satisfy the three listed constraints.

Different patterns are inter-related in a way that one pattern applies other patterns to formally describe certain sub-functions of the intended function. In the pattern *sorting*, for example, pattern *group* is involved in the first rule for generating a part of the resultant formal expression.

4.1.2 Pattern definition

In our approach, the specification patterns are designed to be applied by machines; any ambiguity will impede their automatic utilization. For this reason, we formalize the pattern structure in the following definition where $\mathcal{P}(s)$ denotes the power set of set s.

Definition 1 *A pattern p is a 6-tuple* $(f, E, PR, expl, \Phi, \Psi)$ *where*

- *f is the unique identity of p denoting the kind of function that p is used to formalize*
- *E is a set of elements where each element denotes one of the necessary attributes of f*
- *PR is a set of constraints on p or the elements in E*
- *$expl : \{f\} \cup E \cup PR \longrightarrow string$ informally interprets f, elements in E and constraints in PR for the purpose of human-machine interaction where string denotes the universal set of strings*
- *$\Phi : \Phi_E \cup \Phi_R$ denotes the set of clarification rules for guiding the assignment of the elements in E where*
  - *$\Phi_E : E \longrightarrow E$ is a partial function that determines the order for specifying elements where*
    - *$\exists_{e_0 \in E} \cdot e_0 \notin ran(\Phi_E)$ (e_0 represents the first element to be specified)*
    - *$\forall_{e \to e' \in \Phi_E} \cdot e \neq e'$ ($e \to e'$ denotes a maplet in Φ_E where e' should be specified after e)*
  - *$\Phi_R : E \longrightarrow RPT$ defines a rule repository for each element e in E to guide the assignment of e where each repository in RPT is a triple (CR, R_0, γ) where*
    - *$CR : \mathcal{P}(PR) \longrightarrow \mathcal{P}(PR)$ denotes the set of rules in the repository where each rule determines the satisfaction of a set of constraints based on already satisfied constraints and $\forall_{PR_i \to PR'_i \in CR} \cdot PR_i \cap PR'_i = \varnothing$*
    - *$R_0 \subset CR$ is the first set of candidate rules to be applied*
    - *$\gamma : CR \to \mathcal{P}(CR)$ determines the sequence for applying the rules in CR where $\gamma(r)$ indicates the candidate rules for further clarifying e after rule r is applied*
    - *$\forall_{RS_i \in ran(\gamma)} \cdot \forall_{PR_m \to PR'_m, PR_n \to PR'_n \in RS_i} \cdot \forall_{pr \in PR_m} \cdot pr \Rightarrow \exists_{pr' \in PR_n} \cdot \neg pr'$ (for each $r \in CR$, only one of the candidate rules in $\gamma(r)$ will be activated when formalizing a function using p)*

- $\Psi : \mathcal{P}(PR) \longrightarrow string$ *is a partial function that denotes the set of transformation rules for generating the formal representation of* f *according to the values assigned to the elements where* $\forall_{PR_i \rightarrow s_i, PR_j \rightarrow s_j \in \Psi} \cdot \forall_{pr \in PR_i} \cdot pr \Rightarrow \exists_{pr' \in PR_j} \cdot \neg pr'$ *(only one of the rules in* Ψ *will be activated for the specified elements)*

The above definition organizes the four items of the pattern structure into a tuple and formalizes them into corresponding elements of the tuple. Specifically, item *name* and *explanation* are denoted as f and the mapping originating from f in *expl* respectively. The other mappings in *expl* are designed to explain the semantics of a subset of the formal concepts in the pattern, which enable the production of comprehensible guidance from formal notations. For item *constituents*, its sub-item *elements* is transformed into a set denoted as E in the tuple and the sub-item *rule for guidance* is formalized into rule set Φ. According to the *rule for guidance* item of the original structure, item Φ formally defines how to guide the retrieval of the elements in E by defining Φ_E and Φ_R. Mapping Ψ mathematically defines item *solution*.

We will describe each item in the formal definition in detail sequentially and then give an example pattern to facilitate the understanding of each item. The first item f reveals the functions that the pattern can be used to formalize. The second item E consists of all the necessary attributes to be clarified to formalize f where each attribute is defined as an *element*. The formal definition of *element* is given as follows.

Definition 2 *An element* e *is a triple* $(id, attr, def)$ *where*

- id *denotes the unique identity of* e
- $attr$ *denotes the attribute that* e *refers to*
- $def = d_1 \cup d_2 \cup ... \cup d_n$ $(n \geqslant 1)$ *denotes the value set that* e *ranges over where*
  - $d_1, d_2, ..., d_n$ *classifies the value set into different categories where each* d_i *denotes a* constitute type *of* e *consisting of a kind of values in* def
  - $\forall_{d_i, d_j \in \mathcal{P}(def)} \cdot d_i \cap d_j = \varnothing$

Ranging over a union type def, each element will be assigned with a value of one of the constitute types according to the function under description when writing specific requirements. Therefore, clarifying a function is actually to refine the definitions of its attributes into one of the constitute types and assign a value of that type.

Table 4.2. An overview on basic element types

element type	definition
nil	The type of the element is not determined yet
$strValue$	Universal set of strings
$numValue$	Universal set of numbers
$expValue$	Universal set of formal expressions for representing system variables where each formal expression is written with defined variables and operators
$typeValue$	Universal set of built-in and custom types
$choice$	$\{c_1, ..., c_n\}$ where each c_i is a candidate item
$constraint$	Universal set of constraints on system variables where $constraint(a)$ denotes constraints on a, $constraint(a_1, ..., a_n)$ denotes constraints on the relations between a_1, ..., a_n

All the constitute types for defining elements are divided into two kinds: *atomic type* and *structured type*. The former indicates to specify an element without further decomposition while the latter decomposes an element into a structure where child elements and low-level requirements are combined to specify high-level elements.

Atomic types include seven members as shown in TABLE 4.2 where the first type nil denotes a special state of elements. An element of nil type can be assigned with a value of any element type and needs to be more specifically defined through further clarification. For each element e of atomic type t, definition format is $e : t$.

Structured types are divided into the following categories:

- *Set* type

  An element of *Set* type refers to an unordered collection of distinct child elements. These child elements share the same def item and differentiate from each other by the different values assigned to them. For an element e defined as a set of elements e', its definition format is $e : set\ of\ e'$.

  As previously introduced, attribute *howAttr* represents the way to alter a given system variable. One way to specify the attribute is to describe various kinds of data items to be modified and depict the performed operation for modifying each kind of data item. Therefore, element *how* for denoting *howAttr* has a constitute type of *Set* type where each child element refers to the performance of the operation on one kind of data item.

- *Composite* type

  An element of *Composite* type is described by a set of child elements $\{e_1, e_2, ..., e_n\}$ where each e_i is defined for specifying one aspect of the corresponding attribute. For an element e of composite type with fields $f_1, ..., f_n$, its definition format is: $e : f_1 \times ... \times f_n$. For example, if certain alter function modifies various kinds of data items, its element *how* should be specified as a *Set* where each child element g of element *how* indicates the performance of the operation on one kind of data item. The element g needs to be described from two aspects: the identification of the data items to be modified (denoted as *data*) and the way to modify the data items (denoted as *oper*). Therefore, *Composite* is a constitute type of g with fields *data* and *oper* reflecting two aspects of the corresponding attribute.

- *Option* type

  An element e of *Option* type is described by a set of child elements $\{e'_1, e'_2, ..., e'_m\}$ where each e'_i is selected from a pre-defined child element set $\{e_1, e_2, ..., e_n\}(n \geqslant m)$. Each e_i denotes a possible aspect of the attribute corresponding to e and at least one element should be selected from the set to compose e.

  Suppose a requirement intends to alter a system variable of *mapping* type. If it modifies part of the variable and gives a set of constraints that describe the target maplets to be altered, these constraints, denoted as element c, may cover three aspects of each target maplet: constraints on its domain (denoted as *dom*), constraints on its range (denoted as *ran*) and constraints on the relations between its domain and range (denoted as *dRr*). Thus, element c can be defined as an *option* type composed of at least one aspect chosen from *dom*, *ran* and *dRr*.

- *Req* type

  An element of *Req* type represents an attribute needed to be described by a low-level requirement formalized by another pattern. For example, for each pair of elements *oper* and *data*, *oper* needs to be specified by a lower-level requirement on function *alter* if *data* is intended to be modified by an *alter* operation. Therefore, one of the constitute types of *oper* is *Req*.

For elements of *req* type, three kinds of formats are provided.

- $e : p$: element e will be assigned with a function formalized by applying pattern p
- $e : p(v_1, ..., v_n)$: let the item E of the pattern p be $\{e_1, ..., e_m\}(m \geq n)$, element e will be assigned with a function formalized by applying p with each e_i specified as v_i where $\forall_{e_i, e_{i+1} \in E} \cdot e_{i+1} = \Phi_E(e_i)$.
- $e : c(v_1, ..., v_n)$: element e ranges over a set of formalized functions $\{p_1(v_1, ..., v_n), ..., p_m(v_1, ..., v_n)\}$ where category c consists of m patterns $\{p_1, ..., p_m\}$ (The concept of category will be introduced in the next section).

The elements in the item E should be designed to satisfy the following three properties.

- Each two different individual functions composed of the elements in the same set E have at least one element assigned with different values. That is, elements in the item E should be able to distinguish individual functions with different function details.
- For any subset E' of set E, there exist at least two different individual functions composed of the elements in E' that hold the same value for each element. That is, any subset E' of the set E is not complete for composing an individual function.
- For any parent set E' of the set E, the set of individual functions composed of the elements in E' is the same as the set of individual functions composed of elements in E. That is, the set E is minimal for completely describing the corresponding kind of function.

To achieve more precise description of the above three properties, we describe them in formal notations as follows where R_E denotes all the individual functions that are composed of elements in set E, e_r denotes the element e of the function r and UE denotes the universal set of elements.

Definition 3
- $\forall_{r_i, r_j \in R_E} \cdot (\forall_{e \in E} \cdot e_{r_i} = e_{r_i}) \Rightarrow r_i = r_j$
- $\forall_{E' \in \mathcal{P}(UE)} \cdot E' \subset E \Rightarrow \exists_{r_i, r_j \in R_{E'}} \cdot r_i \neq r_j \land \forall_{e \in E'} \cdot e_{r_i} = e_{r_j}$

- $\forall_{E' \in \mathcal{P}(UE)} \cdot E' \supset E \Rightarrow R_E = R_{E'}$

The third item PR includes three kinds of constraints. The first kind is the propositions on the pattern p. The second kind is *definition constraints*, i.e., the constraints on element definition. The third kind is *value constraints*, i.e., the constraints on element value. All these constraints can be evaluated as either true or false when formalizing certain function using p. Their evaluation results determine the guidance to be displayed and the formal expression of the function.

The fourth item *expl* converts three kinds of objects into their informal explanations for forming comprehensible guidance. The first object is f and $expl(f)$ gives the informal explanation on f so that the developer can obtain a better understanding on the functions that p can be used to formalize. The second kind is the elements included in E. For each element e, $expl(e)$ indicates the attribute of f that e stands for. The third kind is the definition constraints and value constraints in PR. For each definition constraint or value constraint pr on element e, $expl(pr)$ indicates the informal guidance that requires for assigning e with a value satisfying pr.

The fifth item Φ is designed to tackle the first task in formalizing the corresponding function f: clarifying the necessary details of f. Since the elements in E indicate all the attributes needed to be clarified to formalize f, clarifying f is actually to assign appropriate values to these elements according to the intended requirement. Therefore, a set of rules are provided by Φ to guide the assignments of the elements in E. These rules are divided into two groups. The first group Φ_E reveals the order for specifying the elements in E where each rule $e \rightarrow e'$ means that element e should be assigned before e'. The second group Φ_R provides a rule repository for guiding the clarification of each element in E step by step. Each rule repository (R, R_0, γ) owns a rule set R and determines the sequence for applying the rules in R. Each rule in R infers new constraint on the value or definition of certain attribute from premise constraint. It will be activated and applied if the premise constraint can be satisfied. This application results in new constraint serving as a guideline that requires for assigning the relevant element with a value satisfying the new constraint. Assume e and e' are two attributes to be clarified where e' is defined as *union* type $expValue \mid strValue \mid constraint$ and "$dataType(e) = mapping \rightarrow e' : expValue$" is one of the clarification rules in the clarification rule repository of e' ($dataType(e)$ denotes the data type of the system variable assigned to element e). This rule indicates that if the premise constraint "e is assigned a system variable of mapping type" can be satisfied, the new constraint "the definition of e' is refined from the original *union* type into one of the

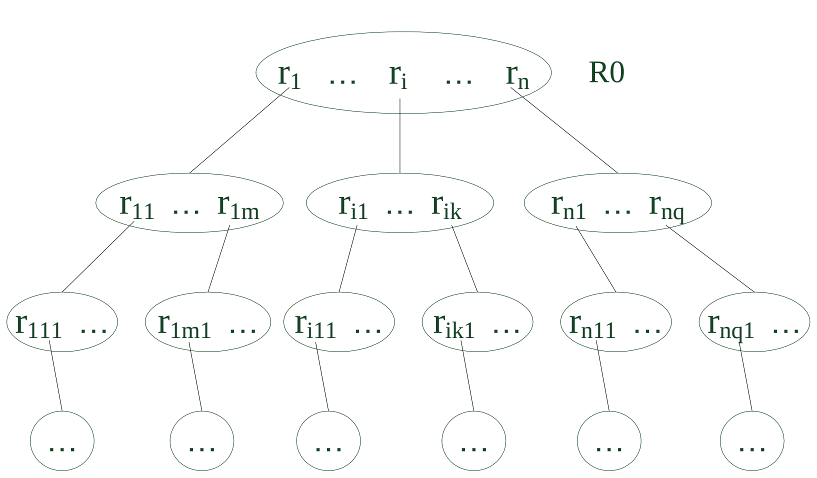


Figure 4.9. The structure of clarification rule repository

constituent types $expValue''$ will be obtained. The obtained new constraint further clarifies the inner structure of e' and serves as a guidance for assigning e'.

For each attribute e_i, the corresponding rule repository organizes all the clarification rules hierarchically as shown in Figure 4.9. Each node in the hierarchy denotes a set of clarification rules with exclusive premise constraints. The root node $R0$ indicates the first rule set to be applied for clarifying e_i and the branches indicate the top-down sequence for applying all the rule sets. Each branch $r \to n$ means that the low-level rule set n is applied after high-level rule r is activated and applied. Let $\{R_{i1}, R_{i2}, ..., R_{im}\}$ be the nodes located at level i in the hierarchy where each R_{ij} contains a set of clarification rules $\{r_{ij1}, ..., r_{ijk}\}$. The branch connecting each rule r_{ijl} in each R_{ij} and a node $R_{(i+1)q}$ located at level $i+1$ indicates that $R_{(i+1)q}$ will be applied if R_{ij} is applied and r_{ijl} is activated when applying R_{ij}.

The sequence for applying the rule repository is $R_0, R_1, ..., R_n$. Each R_i denotes the ith applied set of candidate rules where each candidate rule is a rule in CR and the premise constraints of all the candidate rules in R_i are exclusive. Only one of the candidate rules can be activated when applying R_i. Therefore, applying each candidate rule set R_i is actually applying its activated candidate rule and treating the newly derived constraint as guidance. After R_i is applied and the response to the produced guidance is received, the next candidate rule set R_{i+1} is determined as $\gamma(r_j)$ where r_j denotes the activated rule when applying R_i. If R_{i+1} is not an empty set, it will be applied to guide the further clarification of the corresponding element. Otherwise, the application of the rules in the rule repository is terminated and the clarification of the corresponding element is finished.

The last item Ψ solves the second task in formalizing f: representing the clarified f in formal notations, i.e., generating the formal expression of the clarified f according to the values assigned to the elements in E.

Figure 4.10 shows the formal definition of the previously introduced pattern *sorting* where $dataType(x)$ denotes the data type of the value assigned to element x.

In this formal definition, f is the same as the item *name* of the corresponding pattern structure. Item E is defined as a set comprising the four elements listed in the *elements* item where element $ruleType$ is defined as a sub-attribute of $rule$. There are three elements in the pattern *sorting* and the attributes that these elements stand for are given in the *expl* item of the definition (These attributes will be presented when explaining *expl*). Elements $objs$ and $result$ are both defined as $expValue$ types. The last element $rule$ is of composite type with two fields: $ruleType$ defined as *choice* type with four candidate items and *content* defined as *nil* type meaning its definition cannot be decided at the beginning of the formalization process.

The item PR of the pattern *sorting* involves all three kinds of constraints. The constraint "*Reuse*" belongs to the first kind "propositions on the pattern". It means that the pattern is applied to describe sub-functions for the application of other patterns. The constraint "$ruleType : \{etg, gr\}$" belongs to the second kind *definition constraints*. Element $ruleType$ is initially defined as a *choice* type with four candidate items and this constraint refines the definition by eliminating two of the candidate items. If the constraint establishes, specifying element $ruleType$ will be facilitated as fewer candidate items are provided. The constraints "$ruleType = et$" and "$dataType(objs) = char$" belong to the third kind *value constraints* where the latter means that the value assigned to the element $objs$ is a character.

The item *expl* of the pattern *sorting* also involves all three kinds of mappings described previously. The first mapping belongs to the first kind "explanation on f". It provides an explanation on sort functions, which reflects the *explanation* item of the corresponding pattern structure. Mapping 2, 3, 4 belong to the second kind "explanation on the elements of the pattern". They reveal the attributes denoted by elements $objs$, $result$, and $rule$ respectively. These attributes will replace the corresponding element names in the produced guidance to allow interactions on the semantic level. Mapping 5 belongs to the third kind "explanation on the constraints of the pattern". It generates the informal guidance for a definition constraint on element $rule$. By following this guidance, the developer will specify $rule$ with a value that satisfies the definition constraint. is the definition constraints and

f	sorting	
E	{*objs:* expValue, *result:* expValue, *rule:* ((*ruleType:* {et, etg, r, gr}) × (*content:* nil))}	
PR	{reuse, |*objs*| = 2, dataType(*objs*) = set of nat0, dataType(*objs*) = char, *rule.ruleType* = et, *rule.ruleType*: etg | gr, *rule.content*: constraint, ...}	
expl	sort $\xrightarrow{1}$ "The placement of objects in a particular order" *objs* $\xrightarrow{2}$ "the set of objects to be sorted" *result* $\xrightarrow{3}$ "the sorting result" *rule* $\xrightarrow{4}$ "the intended rule for sorting the given objects" *rule*: (*ruleType*: {et, etg, r, gr} × (*content*: nil)) $\xrightarrow{5}$ "Specify *rule* from two aspects: 1. the category of *rule* 2. the detail of the *rule*. For aspect 1, choose from: each pair of neighbor objects in the sorting result holds the same relation, objects are organized into groups and each pair of neighbor groups in the sorting result holds the same relation, more than one rule is used to sort the objects, more than one rule is used to sort the grouped objects" *gR* $\xrightarrow{6}$ "the intended rule for grouping the given objects before sorting" ……	
Φ	**Φ_E**	{*objs* → *result*, *result* → *rule*}
	Φ_R	*objs* → (R1, {r1}, γ) R1: {true} $\xrightarrow{r1}$ {*objs* : expValue} {dataType(*objs*) = char} $\xrightarrow{r2}$ {re(*objs*)} {dataType(*objs*) = set} $\xrightarrow{r3}$ ∅ …… γ : r1 → {r2, r3, ⋯}, r2 → {r2, r3, ⋯}, r3 → ∅, ⋯ *result* → (R2, {r4}, γ) R2: {true} $\xrightarrow{r4}$ {*result* : expValue} {dataType(*result*)=seq of dataType(*objs*)} $\xrightarrow{r5}$ ∅ …… γ : r4 → {r5, ⋯}, r5 → {r5, ⋯}, r5 → ∅, ⋯ *rule* → (R3, {r6, r7, ⋯}, γ) R3: {|*objs*| = 2} $\xrightarrow{r6}$ {*rule.ruleType* = et} {dataType(*result*) = seq of dataType(*objs*)} $\xrightarrow{r7}$ {*rule.ruleType*: {etg, gr}} {*ruleType* = et, dataType(*objs*) = mapping} $\xrightarrow{r8}$ {*rule.content*: Relation(domi, domj) × Relation(rngi,rngj) × Relation(domi,rngj)} {*ruleType* = etg} $\xrightarrow{r9}$ {*rule.content*: (*gR*: group(*objs*, elems(*result*)) × (*sR*: nil)) {dataType(*objs*) = set of composed of f1, ⋯, fn, dataType(elems(*result*)) = set of dataType(*objs*), *rule.content.gR.grule.content* ∈ {f1, ⋯, fn}} $\xrightarrow{r10}$ {*rule.content.sR*: constraint($\|g_i\|$, $\|g_{i+1}\|$) | constraint(g_i.(*gR.grule.content*), g_{i+1}.(*gR.grule.content*)) | ⋯ } …… //*gR.grule.content* denotes the value assigned to low-level element *grule.content* when applying the pattern *group* for clarifying element *gR*. γ : r6 → {r8, ⋯}, r7 → {r9, ⋯}, r9 → {r10,⋯}, r10 → ∅, ⋯
Ψ	{!reuse, dataType(*objs*) = set of composed of f1, ⋯, fn, dataType(*result*) = seq of dataType(*objs*), *rule.ruleType* = etg, *rule.content*: constraint($\| g_i \|$, $\| g_{i+1} \|$)} $\xrightarrow{tr1}$ "*rule.gR* and forall[g, g' : elems(*result*), i : int] | elems(*result*)[i] = g elems(*result*)[j] = g' => card(g) > card(g')" ……	

Figure 4.10. The formal definition of the pattern "sorting"

value constraints in PR.

Item Φ is composed of Φ_E and Φ_R. In the item Φ_E of the pattern *sorting*, rule $objs \rightarrow result$ means that the element $objs$ is first specified and $result$ should be specified after $objs$. The other rule $result \rightarrow rule$ indicates that element $rule$ is required to be specified after $result$ is specified.

In the item Φ_R of the pattern *sorting*, the rule repositories of the three elements are given. The rule repository for $objs$ is $(R1, \{r1\}, \gamma)$ where three example rules $r1$, $r2$, $r3$ in $R1$ are listed and $\{r1\}$ is the first candidate rule set to be applied. Premise constraint *true* indicates that $r1$ will be activated under any condition. Therefore, the application of $\{r1\}$ is actually the application of $r1$ which results in the constraint "$objs$: $expValue$". This constraint serves as a guideline that asks for the assignment of $objs$ with a value of $expValue$ type. When the response from the developer is received, candidate rule set $\gamma(r1)$, i.e., $\{r2, r3, ...\}$, will be applied by applying its activated rule. Assigning different values to $objs$ leads to different activated rules.

For example, if the given $objs$ is a character, $r2$ will be activated and applied where the retrieved constraint $re(objs)$ means that $objs$ should be assigned again with a new value since sorting one character makes no sense (We use the keyword re in pattern definitions to represent reassignment. For each element e, $re(e)$ means the reassignment of e). But if the given $objs$ is a set, $r3$ will be activated and applied where $\varnothing$ means that $objs$ needs not to be clarified further at the moment. Assume ar is the activated rule when applying $\gamma(r1)$, the next candidate rule set to be applied after the application of $\gamma(r1)$ is $\gamma(ar)$. Such kind of process repeats until the application of a final rule fr where $\gamma(fr) = \varnothing$. We skip the explanation on the rule repository of element $result$ since it is similar to that of $objs$. In the rule repository for element $rule$, the first candidate rule set $R3$ includes more than one candidate rules for initial clarification of element $rule$. Rules $r6$ and $r7$ are given as the example rules in $R3$. Rule $r6$ states that if the given $objs$ owns two members, composite element *rule*'s field *ruleType* will be assigned as *et*. Rule $r7$ means that if its premise constraint can be satisfied, the definition of *rule*'s field *ruleType* will be refined from four to two candidate items. The sequence for applying other candidate rule sets after $R3$ is determined by the item γ. For example, $\{r8, ...\}$ will be applied if $r6$ is activated when applying $R3$ and $\{r9, ...\}$ will be applied if $r7$ is activated.

Note that the constraints derived by $r8$ and $r9$ in the pattern *sorting* involve the use of categories and patterns such as *Relation* and *group*. They utilize the definition format for defining elements of *req* type. For example,

"$group(objs, elems(result))$" is used to define low-level element gR in $r9$ (According to mapping 6 in the *expl* of the pattern *sorting*, gR refers to attribute "the intended rule for grouping the given objects before sorting". It is created and defined as a low-level element for specifying one aspect of the high-level element *rule.content*). It indicates that gR will be assigned as the formal representation generated by applying the pattern *group* with the element information $(objs, elems(result))$. As shown in Figure 4.11, three elements are included in the item E of the pattern *group*. The Φ_E item reveals that $gobjs$ and $gresult$ are the first two elements to be specified. Therefore, the above element information means that when applying the pattern *group* for assigning gR, elements $gobjs$ and $gresult$ are assigned as $objs$ and $elems(result)$ respectively.

Item Ψ generates formalization results for sorting functions according to the values assigned to the three elements. In some formalization results, element names are used to denote the values assigned to the corresponding elements when applying the pattern. For example, rule $tr1$ in the pattern *sorting* has five premise constraints. If all these five constraints can be satisfied by the values assigned to the three elements, the formal expression starting with "$rule.gR$" will be generated as the suggested formal representation of the clarified f.

On the basis of the above formal definition, individual patterns can be created. Different patterns have different elements in the E item and different rules in the Φ and Ψ items. Since their structures are consistent with the formal definition, their complexities are the same as the complexity of the definition. Due to the fact that each individual pattern needs to be designed by analyzing the semantics of the corresponding kind of function, the construction of patterns is an intellectual work without a general method. It is manually done according to our experience and understanding on the semantics of the corresponding functions.

4.2 The hierarchy in the pattern system

Due to the inherent complexity of software, the number of patterns will be so large and the pattern users will find that the selection of an appropriate pattern becomes a hard task. Moreover, the management of these large number of disordered patterns will be difficult. As more and more new patterns are introduced in, selecting and managing would become more and more complicated. To overcome these drawbacks, we divide them into distinct categories and organize them in a hierarchical structure in the pattern system by categorizing the functions they are used to formalize.

f		group
E		{*gobjs:* expValue, *gresult:* expValue, *grule:* (*ruleType:* {uR, iR} × (*content:* nil))}
PR		{reuse, $\forall$ obj_i, obj_j ∈ *gobjs* · dataType(obj_i) = dataType(obj_j), ...}
expl		group → "Dividing a collection of objects into groups" *gobjs* → "the objects to be divided" *gresult* → "the grouping result" *grule* → "the rule for dividing the objects" *grule*: (*ruleType*: {uR, iR} × (*content*: nil)) → "Specify *grule* from two aspects: 1. the category of *grule* 2. the detail of *grule*. For aspect 1, choose from: certain parts of the objects in each group are the same, objects in each group satisfy the same properties" ……
Φ	**Φ_E**	{*gobjs* → *gresult*, *gresult* → *grule*}
	Φ_R	*gobjs* → (R' 1, {r1}, γ) R' 1: {true} $\xrightarrow{r1}$ {*gobjs* : expValue} {dataType(*gobjs*) = set} $\xrightarrow{r2}$ ∅ …… γ : r1 → {r2, ⋯}, ⋯ *gresult* → (R' 2, {r3}, γ) R' 2: {true} $\xrightarrow{r3}$ {*gresult* : expValue} {dataType(*gresult*) = set of dataType(*gobjs*)} $\xrightarrow{r4}$ ∅ …… γ : r3 → {r4, ⋯}, ⋯ *grule* → (R' 3, {r5}, γ) R' 3: {true} $\xrightarrow{r5}$ *grule.ruleType*: {uR, iR} {dataType(*gresult*) = set of dataType(*gobjs*), dataType(*gobjs*) = set of composed of f1, ⋯, fn, *grule.ruleType* = uR} $\xrightarrow{r6}$ {*grule.content*: {f1, ⋯, fn}} …… γ : r5 → {r6, ⋯}, r6 → ∅, ⋯
Ψ		{reuse, dataType(*gresult*) = set of dataType(*gobjs*), $\forall$ obj_i, obj_j ∈ *gobjs* • dataType(obj_i) = dataType(obj_j), dataType(*objs*) = composite, *rule.ruleType* = uR} $\xrightarrow{tr1}$ "forall[g: *gresult*] | (forall[t: g] | t inset *gobjs*) and (forall[ti, tj: g] | ti.(*grule.content*) = tj.(*grule.content*)) and forall[gi,gj: *gresult*] | not exist[t: gi, t' : gj] | t.(*grule.content*) = t' .(*grule.content*) and forall[t: *gobjs*] | exists[g: *gresult*] | t inset g" ……

Figure 4.11. The formal definition of the pattern "group"

Figure 4.12 shows the hierarchy where each rectangle denotes a category and each oval denotes a concrete pattern. The child nodes of each node n denote the subcategories for composing category n or the patterns for composing category n. The root node denotes the pattern system and its child nodes denotes the top-level categories for classifying all the patterns. As can be seen from the figure, there are two top-level categories: Unit Function denoted as UF and Compound Function denoted as CF. This indicates that all the patterns can be divided into two categories: one for describing unit functions and the other for depicting compound functions. Their sub-categories are further classified into more specific sub-categories or patterns.

Category CF includes the concrete patterns for formalizing compound functions such as conditional function $if - then - else$ and multiple choice function $case$. With the experiences from many typical formal specifications, we found that most of the functions are described by the combination of three kinds of basic functions: relations between objects, acquisition of information and updating of existing data. Therefore, category UF is divided into three sub-categories: *Relation* patterns, *Retrieval* patterns and *Recreation* patterns.

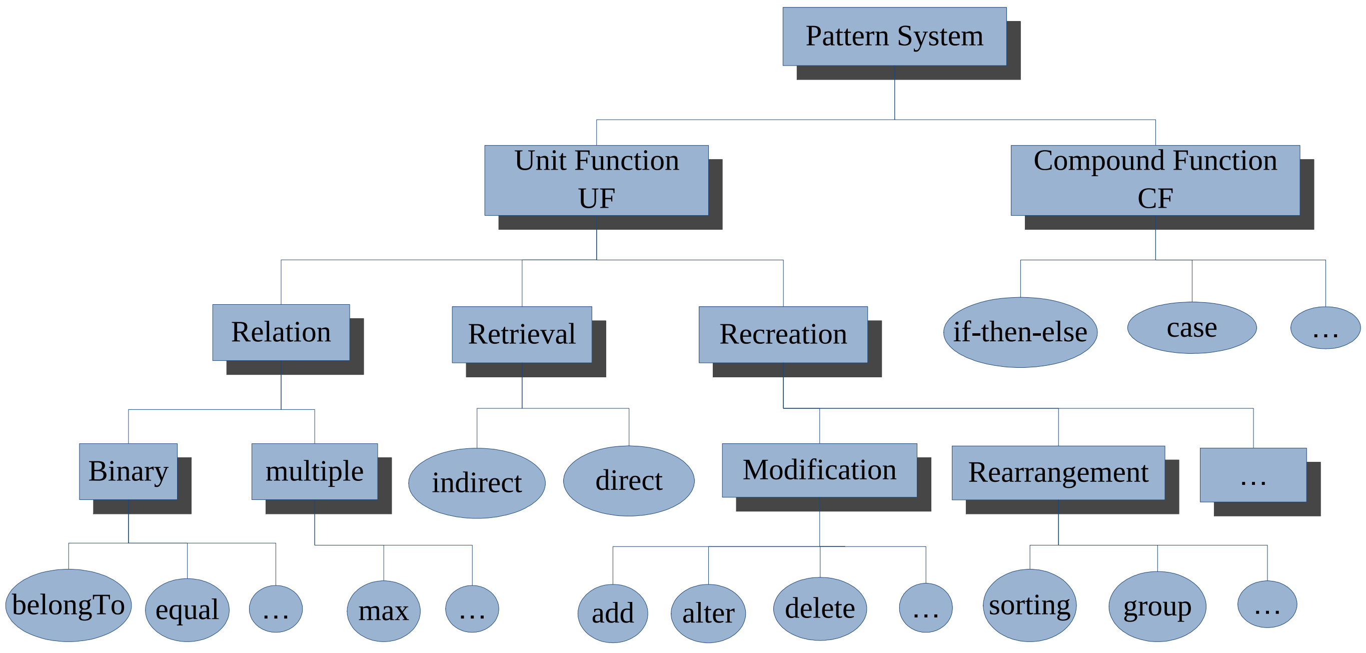


Figure 4.12. Pattern categorization

Relation patterns deal with formal descriptions of relationships between objects, such as the equivalence of two variables. They will not cause any changes on the state of the system and always yield a boolean value as their result to indicate whether the relations are satisfied. There are two kinds of relation patterns: Binary patterns and multiple patterns. The former is used to formalize the description of relations between two objects while the latter is usually used to formalize the relations between objects of more than two. For example, *belongTo* pattern formalizes the binary relation where certain object belongs to another and the pattern *max* is often describe the comparison among multiple objects.

Although some of the system variables are already defined in formal specifications, but vast majority of them need to be represented by combinations of defined variables, such as the balance of one's account in a banking system. And retrieval patterns are designed to generate expressions that show the meaning of obtaining data items from a compound data structure of any kind of compound types (e.g., set, sequence, map). The data items can be values of either basic types (e.g., integers, real numbers) or compound types (e.g., sets, sequences). These values are usually represented by variables of specifications that include not only explicit variables explicitly defined and used in the specifications, but also implicit variables presented as combinations of explicit variables. For a specification that involves variables with complex relations, it is difficult to figure out such combination for certain required implicit variables. To make it easier, retrieval patterns are used to gradually retrieve the implicit variables from explicit variables by means of interactions with users. In most cases, the retrieval patterns need to apply themselves for further formalization and gradually reach the goal data items. There are two patterns in the retrieval category: indirect and direct patterns. The former deals with the functions that obtain a system variable by operating on other variables while the latter deals with the functions that obtain the data items within one variable.

Recreation patterns offer solutions to formally presented changes on the state of the system. A formal specification describes the state change of the system by expressions that define the final state variables based on the initial state variables. The main task of recreation patterns is to help users specify the objects they want to modify and achieve the target new values. This category can be further classified into sub-categories. For example, sub-category modification includes the patterns for formalizing the function of general update operations on system variables where the pattern *add* deals with the function of adding new data items to existing variables, the pattern *alter* deals with the function of altering the data items of the existing variables and the pattern *delete* deals with the deletion of data items in existing variables. Another sub-category rearrangement is composed of all the patterns for formalizing the functions of changing the organization mechanism of a set of objects. For example, the pattern *sorting* deals with the changing of the order of a sequence and the pattern *group* deals with the changing of the division of a set of objects.

Currently, there are 41 patterns in the pattern system where 31 patterns are included in category *UF* and 10 patterns in category *CF*. Among the patterns in *UF*, 9 of them compose sub-category *Relation*, 2 of them compose sub-category *Retrieval* and 20 of them compose sub-category *Recreation*. Patterns in *Relation* and *Recreation* are divided into two and five categories respectively.

We also provide a formal definition for the pattern system with the above hierarchy that categories all the patterns.

Definition 4 *Pattern System is a triple (P, C, η), where P denotes the universal set of patterns, C is a set of categories for classifying the patterns in P, $\eta : C \longrightarrow \mathcal{P}(C \cup P)$ determines the sub-categories or patterns staying one level lower than each category $c \in C$ where $\exists_{c_0 \in C} \cdot c_0 \notin ran(\eta)$ (c_0 is the root of the pattern system).*

The final goal of the pattern system is to support the requirements formalization for general system development, i.e., the formalization of most of the commonly used functions for composing the requirements of all kinds of software systems. But as the first step, the domain of the current pattern system described in this thesis is the bottom level functions that are commonly used to compose more complex functions in typical formal specifications. Therefore, the pattern system is not expected to support the formalization of high-level functions or domain-specific functions, such as the *money transfer* we mentioned previously. The formalization of these kinds of functions need domain-specific knowledge to guide their decomposition into a set of bottom-level functions that can be formalized using our pattern system. Therefore, the current pattern system requires the developers to use their own domain-specific knowledge to analyze the high-level functions and obtain the involved bottom-level functions themselves.

Besides, the pattern system is not expected to be complete since all the patterns actually form a pattern language for describing requirements and proving the expressive of a language is almost impossible. It will be constantly updated to enable the formalization of bottom-level functions of wider range. During the use of the pattern system, we may find some bottom-level functions that cannot be formalized by the available patterns, then new patterns and categories will be designed and introduced, or existing ones will be modified to provide more efficient guidance. Such a task is accomplished manually at present and its automation would be gradually enhanced as further research progresses are made.

4.3 Summary

In this chapter, we have explicitly described the *specification pattern system* which is the fundamental concept of our requirements formalization approach. The description is started from the concept of *specification pattern.* Instead of directly giving the formal definition of the pattern, we first explain its underlying theory and structure so that the reader is able to set a firm foundation for the understanding of the formal notation. After presenting pattern's formal definition, an example pattern is given to illustrate each component of the definition. We also show the hierarchy of the pattern system where all the patterns are classified according to the functions they can be used to formalize.

In the next chapter, we will describe how the *specification pattern system* is applied to guide the requirements formalization process. The application includes two activities. The first activity is to the main stream of refinement from informal requirements to formal specifications. The second activity is the declaration of data types that are necessary for performing the first activity. We will describe these two activities in detail respectively.

Chapter 5

Requirements formalization based on the specification pattern system

The major task in requirements formalization is to describe software behaviors in formal expressions such as pre- and post-conditions of operations. Another important task is the declaration of data types since writing formal expressions requires the availability of a set of state variables which needs to be formally defined by data types. Instead of performing sequentially, these two tasks are usually interleavingly carried out. For each function to be formalized, if the existing data types are not sufficient or inappropriate for formally describing it, the developer will be guided to create new types or modify the existing ones according to the need of the function. Then the description of the function continues with the updated data types and the declaration activity will be repeatedly performed to deal with the later encountered data type problems.

We will describe the methods for supporting the above two activities in detail, respectively. For function description in formal expressions, we assume the necessary data types are defined by applying the data type declaration method and focus on explaining the generation process of the target formal expressions from informal requirements. For data type declaration, we will show how the functions to be formalized guide the defining of the appropriate data types.

5.1 From informal requirements to formal specifications

Well-defined pattern structure and pattern system hierarchy establishes a firm foundation for guiding the requirements formalization process. Applying the *specification pattern system* to refine informal requirements into formal specifications largely depends on pattern structures and is therefore straightforward. It consists of two steps.

The first step is to select an appropriate patterns from the pattern system for guiding the formalization of the intended requirements. The second step is to apply each selected pattern to obtain the target formal expressions.

Specifically, given a requirement rq, its formalization based on the pattern system contains two major steps.

Step 1 Pattern selection

Appropriate patterns for formalizing rq need to be selected first. The selection process can be guided by the hierarchy of the pattern categorization. Starting from the top level of the hierarchy, the developer is required to select a sub-category on each level until reaching a pattern p. It is not difficult to find the right pattern because of three reasons. First, pattern names are written in natural language and designed to be distinguishable from each other on the semantic level. Second, the patterns are organized by categories at different levels and the developer only needs to deal with one category or sub-category at a time. Third, the *expl* items of the patterns describe their usage in more details and can help confirm the selection decision.

In most cases, it is hard to find a pattern specifically designed for formalizing rq if rq is a high-level function consisting of a set of basic functions since the current patterns are designed to deal with basic functions. Human intelligence is needed to analyze rq on the semantic level and decompose it into a set of basic functions where each basic function can be formalized by a pattern. For example, when formalizing the $money\ transfer$ function for an ATM system, one cannot find a pattern specifically designed for formalizing $money\ transfer$ functions. Considering that the function's meaning is to transfer certain amount of money from an original account to a destination account, it can be divided into two sub-functions: add the amount of the transferred money to the destination account and delete the same amount of money from the original account. These two sub-functions are essentially updating system variables and therefore can be formalized by selecting patterns from recreation category.

Step 2 Pattern application

With a set of patterns $\{p_1, ..., p_n\}$ selected for all the sub-functions $\{ f_1, ..., f_n\}$ of rq, the next step is to apply them. Each pattern p_i denoted as $(f, E, PR, expl, \Phi, \Psi)$ is applied by the following two steps.

(a) requirement clarification

Based on the specification patterns, requirements clarification is to instantiate the appropriate specification patterns by specifying the relevant elements. It generates requirements composed of elements assigned with concrete values. The assignment of these elements is guided according to their definitions and results in clarified requirements where all the elements are specified with values. The formal definitions of the involved elements guarantee the accuracy of the requirement so that it can be automatically transformed into formal specifications.

In this step, the developer will be guided to clarify the necessary details of f_i by assigning values to the elements of p_i. Specific algorithm is as follows where e_0 denotes the first element to be specified when applying p_i.

```
$ce = e_0$; // $ce$ denotes the element being specified
$while$($ce$ is an element in $E$){
  $if$($ce$ has not been specified){
    retrieve the rule repository $\Phi_R(ce) = (CR, R0, \gamma)$;
    $CRS = R0$; //$CRS$ denotes the candidate rule set being applied
    while($CRS \neq \varnothing$){
      apply $CRS$ by applying its activated rule $ar$;
      accordingly display guidance and receive response;
      $CRS = \gamma(ar)$;
    }
  }
$ce = \Phi_E(ce)$;
}
```

(b) formal expression generation

In this step, an expression exp that formally describes f_i will be generated based on the values assigned to the elements in p_i. Specific algorithm is as follows.

```
for each ($PR_i \rightarrow str$)$\in \Psi$
```

$$if(\forall_{pr \in PR_i} \cdot pr) \qquad then \;\; exp = \Psi(PR_i);$$

It should be noted that there may exist informal expressions in exp, such as the previously mentioned group statement. These informal expressions should be further formalized by applying the relevant patterns through the above two steps.

A case study on the function *transaction analysis* of the previously introduced banking system is presented to illustrate the above two steps. Figure 5.13 shows the types and variables formally defined for writing the formal specification of the example function *transaction analysis* (Since this part focuses on the presentation of the transformation from informal requirements to formal specifications, it is assumed that the relevant types and variables are already defined. How to define appropriate data types will be explained in the next section).

```
type
  AccountNo = seq of nat0;
  Password = string;
  CustomerInf = composed of
                   accountNo: AccountNo
                   password: Password
                  end;
  CurrencyType = {<USD>, <JPY>, <CNY>};
  Amount = real;
  Balance = map CurrencyType to Amount
  Year = nat0;
  Month = nat0;
  Day = nat0;
  Date = Year*Month*Day;
  OperationType = {<deposit>, <withdraw>};
```

```
 Transaction = composed of
                   date: Date
                   operationType: OperationType
                   currencyType: CurrencyType
                   amount: Amount
                  end;
  AccountInf = composed of
                  balance: Balance
                  transactions: set of Transaction
                 end;
  AccountFile = map CustomerInf to AccountInf;

var
  ext #account_store: AccountFile;
  ext #today: Date;
```

Figure 5.13. Types and variables declared for the banking system

The example function *transaction analysis* extracts transactions from certain account and sorts them by dividing them into different groups by date and sorting these groups by the number of the included members in descending order. According to the CDFD of the banking system, process $Tran\_Analysis$ requires for two inputs: input $inf1$ denoting that the manager owns a valid ID, input $inf : CustomerInf$ denoting the customer owning the account that the desired transactions belong to. It produces an output $tranList : seq\ of\ set\ of\ Transaction$ according to the input which denotes the desired transaction list.

With the well-defined input and output, the formalization of the function *transaction analysis*, i.e., the generation of the post-condition of the process $Tran\_Analysis$, can be started and performed according to the two steps in requirements formalization.

Step 1 Pattern selection

The hierarchy of the pattern categorization is first shown for pattern selection. By analyzing the semantics of the example function, the hierarchy is traversed from the top level to an appropriate pattern through the path: $UF \rightarrow Recreation \rightarrow rearrangement \rightarrow sorting$. After confirming the select decision by reading $expl(sorting)$, the pattern *sorting* (previously shown in Figure 4.10) is finally selected to assist the formalization of the example function.

Step 2 The application of the pattern *sorting*

(a) requirement clarification

According to the given algorithm, $objs$ is the first element to be specified in the pattern *sorting* and its rule repository $\{R1, \{r1\}, \gamma\}$ is derived to guide its assignment. As the first candidate rule set, $\{r1\}$ is first applied. Since the premise constraint of $r1$ is *true*, it is activated and applied resulting in a new constraint "$objs : expValue$". By applying $expl(objs)$ and $expl(objs : expValue)$, a comprehensible guidance "Specify the set of objects to be sorted by a system variable described in defined variable or formal expression." is produced from the new constraint. Representing the transactions to be sorted, "$account\_store(inf).transactions$" is given as response to the above guidance and assigned to $objs$ (If the developer finds it hard to write this formal expression, he can apply the pattern *direct* which is used to retrieve the formal representation of system variables). After the application of $r1$, $\gamma(r1) = \{r2, r3, ...\}$ becomes the next candidate rule set to be applied. By automatically evaluating the premise constraint for each included candidate rule, $r3$ is determined as the activated one since its premise constraint can be satisfied by the given $objs$. The derived new constraint is empty and no guidance is provided. Since $\gamma(r3) = \varnothing$, the assignment of $objs$ is thus finished.

The Φ_E item of the pattern *sorting* tells that $\Phi_E(objs) = result$, i.e., element $result$ is the next element to be specified. Again, the rule repository $\{R1, \{r4\}, \gamma\}$ is derived to guide the assignment of $result$. Similar to $objs$, candidate rule $r4$ is first applied resulting in the constraint "$result : expValue$". Accordingly, guidance "Specify the sorting result by a system variable described in defined variable or formal expression." is displayed for a reply. Obviously, output variable $tranList$ is the correct response and used to assign $result$. Candidate rule set $\gamma(r4) = \{r5, ...\}$ is then applied where $r5$ is activated. The derived new constraint is empty and the assignment of $result$ terminates since $\gamma(r5) = \varnothing$.

Element $rule$ is the last element to be specified. According to its rule repository $\{R3, \{r6, r7, ...\}, \gamma\}$, $\{r6, r7, ...\}$ is first applied where $r7$ is activated. Based on the $expl$ item, the derived constraint "$rule.ruleType : \{etg, gr\}$" is transformed into guidance "Specify the category of the intended sorting rule by choosing from: 1. objects are organized into groups and each pair of neighbor groups in the sorting result holds the same relation 2. more than one rule is used to sort the grouped objects". After analyzing the semantics of the example function, the first choice is selected as response and assigned to the $ruleType$ field of $rule$. The next candidate rule set is $\gamma(r7) = \{r9, ...\}$ where $r9$ is activated and applied. The obtained constraint decomposes $rule.content$ into two low-level elements gR and sR. Its meaning is displayed as the guidance "Specify the detail of the sorting rule from two aspects: how to group transactions and how to sort grouped transactions.". According to gR's definition given in the constraint, its assignment is guided by applying the pattern $group$ with the first two elements assigned as $objs$ and $elems(result)$ respectively (In SOFL, $elems(seq)$ means the set of elements in sequence seq).

The application of the pattern $group$ also follows the given two steps. Step a starts from the assignment of element $grule$, since the first two elements $gobjs$ and $gresult$ have already been assigned in gR's definition. According to the pattern, the rule repository of $grule$ is $(R'3, \{r5\}, \gamma)$ and $r5$ is first applied resulting in constraint "$grule.ruleType : \{uR, iR\}$". Based on the $expl$ item, this constraint is displayed as guidance "Specify the category of the intended grouping rule by choosing from: 1. certain parts of the objects in each group are the same 2. objects in each group satisfy the same properties.". In the example function, transactions are grouped according to their date, which belongs to the first kind. Thus, uR is chosen and assigned to $grule.ruleType$. The assignment of $grule$ continues with the candidate rule set $\gamma(r5) = \{r6, ...\}$ where $r6$ is activated. The derived constraint allows the assignment of $grule.content$ by choosing from $f1, ..., fn$ where each fi denotes grouping $gobjs$ according to the field fi. In our case, each fi is instantiated as one of the fields of transaction and the $grule.content$ is guided to be assigned by choosing from *date*, *operationType*, *currencyType* and *amount*. It is easy to make the selection decision according to the semantics of the example function. Item *date* is selected as the response and assigned to *grule.content*. Step a for applying the pattern *group* is then finished since $\gamma(r6) = \varnothing$. Then the application moves to step b with the assigned elements. By automatically evaluating the premise constraint of each rule in the Ψ item of the pattern *group*, rule $tr1$ is activated since its premise constraint can be satisfied by the values assigned to the three elements. The formal expression suggested by $tr1$ is generated as the result of applying the pattern *group*.

Since the above application of the pattern *group* is to clarify the low-level element gR, the generated formal expression is assigned to gR and the algorithm goes back to the application of the pattern *sorting* where $r9$ has been activated and applied. According to $\gamma(r9)$, candidate rule set $\{r10, ...\}$ is then applied where $r10$ is activated. The application of $r10$ results in a constraint on the definition of the low-level element sR which is interpreted, by the *expl* item, as guidance "Specify the detail of the sorting rule by specifying one of the following relations between each pair of neighbor groups (g_i, g_j) in the sorting result: 1. relation between the date of the transactions in g_i and g_j 2. relation between the member numbers n_i and n_j of g_i and g_j 3. $\cdots$". In the example function, transaction groups are sorted by the number of their included members in descending order. Therefore, the sorting rule should be clarified by the second kind of relation and $n_i > n_j$ is input as the response to the above guidance. This response is assigned to low-level element sR and the step a of the application of the pattern *sorting* is finished since $\gamma(r10) = \varnothing$.

(b) formal expression generation

All the three elements are assigned with determined values through step a, that is, all the necessary details of the function *transaction analysis* has been clarified and recorded through step a. These values will be used for the automatic generation of the corresponding formal expression in this step.

Each rule in the Ψ item of the pattern *sorting* is automatically analyzed in the context of the clarified elements to explore the activated one "$\{pr_1, ..., pr_n\} \rightarrow str$" where constraints $pr_1, ..., pr_n$ are all satisfied. String *str* is then generated as the formal representation of the function *transaction analysis*. Since the premise constraint of the rule $tr1$ can be satisfied, $tr1$ is applied and the corresponding formal expression is given as shown in Figure 5.14.

```
forall[g: elems(tranList)] | (forall[t: g] | t inset account_store(inf).transactions)
                                   and (forall[ti, tj: g] | ti.date = tj.date)
 and forall[gi,gj: elems(tranList)] | not exists[t: gi, t': gj] |  t.date = t'.date
 and forall[t: account_store(inf).transactions] |
        exists[g:  elems(tranList)] | t inset g
 and forall[g, g': elems(tranList), i : int] |
        elems(tranList)[i] = g and elems(tranList)[i+1] = g' =>  card(g) > card(g')
```

Figure 5.14. The formal representation of the function *transaction analysis*

The case study simulates the interaction process for guiding the formalization of the function *transaction analysis* based on the pattern system. It demonstrates that the developer will not be aware of the existence of the pattern system when it is applied to guide requirements formalization. Instead of studying and utilizing the pattern system manually, the developer only needs to respond in a specified format to the sequentially displayed guidelines written in natural language. And the formal representation of the intended function is automatically generated based on the collected responses and the pattern system. For any requirement within the scope of the pattern system, practitioners without formal notation expertise are able to formalize it through the application of the pattern system similar to the application process presented in the case study. But for requirements beyond the pattern system's scope, no appropriate patterns can be applied to formalize them. We will keep creating new patterns to expand the application domain of the pattern system.

We give the details of the function *transaction analysis* in advance to enable the explanation of the application process. However, in most real cases, the developer may not be able to have all the necessary details of a function in mind before formalizing it. The provided guidelines reveals what kind of attributes are needed to formalize the intended function and the responses to these guidelines will be adequate to form the formal representation of the function. Besides, the developer can obtain a better understanding on the envisioned system through the interaction process for clarifying the required attributes.

5.2 Data type declaration

As the complexity of software grows, data type declaration becomes more difficult to manage and more likely to result in defected data types. There are mainly two kinds of methods for supporting data type declaration. One is to allow the developers to design data types using graphical notations, such as entity-relation-diagram, and transforms the diagram into formal definitions. It facilitates the declaration process by the use of more intuitive languages. The other is to detect the syntactic errors in the declared data type definitions. It focuses on the correctness of the type information for formal specification construction. However, both methods treat data type declaration as an activity carried out independent from function descriptions. They rarely consider the impact of system behaviors on shaping the data types to be declared. When encountering systems with complex data structures and functionality, designing data types that are appropriate for all functions become rather difficult. Furthermore, neither of the two methods provides guidance on data type declaration. The first method provides a better language without the guidance on how to define data types using the language. The second method only works on the syntactical level when the declaration activity is finished. Even if the developers manage to formally define the initial data types, they will be forced to frequently modify the written formal expressions. The reason is that the initial data types are usually defined according to the first function to be formalized and they often need to be updated to enable the formal descriptions of the following functions. Each time when the type information is updated, the formal expressions written with the original type information must be updated to guarantee their consistency with the data types.

To deal with the above problems, we put forward an approach to supporting data type declarations for requirements formalization based on specification patterns. Its underlying principle is that types should be defined to meet the need of correctly and concisely describing relevant functions. Type definitions will evolve as function description proceeds until all the expected functions are properly represented in formal expressions. During the application of each pattern , necessary data types can be automatically recognized and their definitions will be refined. Specifically, when applying each selected pattern, we use *function-related declaration* to guide the refinement of the related data types. It consists of two steps for different stages of the application process: *property-guided declaration* and *priority-guided declaration.*

We also give a method for updating formal expressions when their involved data types are refined to keep the consistency. When a type definition is modified after the application of a pattern, the formal expressions affected by such modification will be fully explored. For each formal expression, the method first retrieves the pattern applied for writing it and the application process of the pattern. Based on the retrieved information and the modified type definition., the formal expression is automatically updated.

This section is described in two parts. The first part gives the outline of the data type declaration approach. The second part uses several sub-sections to introduce the critical techniques used in the approach.

5.2.1 Approach outline

The proposed approach regards data type declaration as an evolution process along with the writing of formal expressions based on the *specification pattern system*. This evolution process starts with a modulized formal specification and terminates when the detailed behavior of each module is precisely given. Figure 5.15 shows the outline of the approach where x-axis and y-axis indicate the formal specification construction process and pattern application process respectively.

On the assumption that specification architecture is already established where modules are organized in a hierarchical structure and processes of each module are connected by their interfaces, developers will first be required to manually declare data types for defining these interfaces. Since process behaviors is not considered in this stage, the declared data types only reflects the initial idea of the intended functions and will be refined as the function details are clarified.

Then the description of individual processes is started where each process should be attached with a pair of pre- and post-condition. For each pre-/post-condition, a pattern suitable for describing the expected function will first be selected. The selected pattern is then applied. Step 1 is to guide the specifying of its elements and step 2 is to generate an intermediate formal result based on the specified elements. During these two steps, *function-related declaration* is carried out to declare new types and refine the existing type definitions where *property-guided declaration* is carried out on step 1 and *priority-guided declaration* is carried out on step 2. The former guides the refinement of type definitions under the principle that all the properties inferred from the specified elements should be satisfied while the latter provides suggested definition of certain types according to the priority attribute

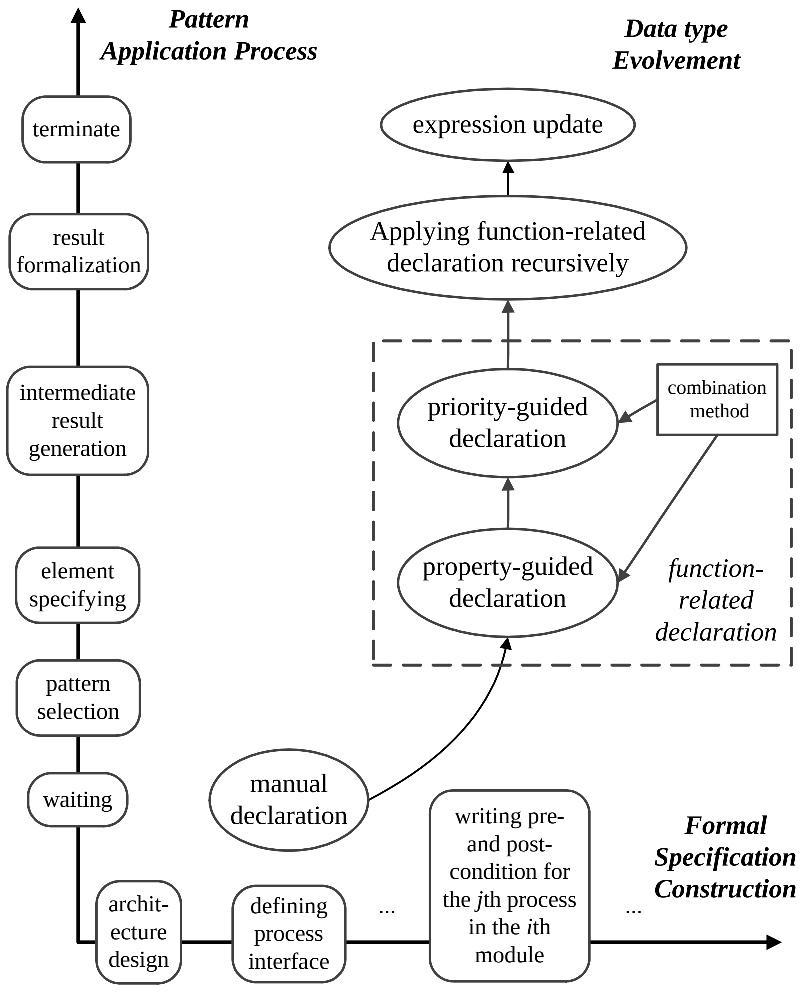

Pattern Application Process
Data type Evolvement
terminate
result formalization
intermediate result generation
element specifying
pattern selection
waiting
expression update
Applying function-related declaration recursively
priority-guided declaration
combination method
property-guided declaration
function-related declaration
manual declaration
archit-ecture design
defining process interface
...
writing pre- and post-condition for the jth process in the ith module
...
Formal Specification Construction

Figure 5.15. The outline of the data type declaration approach

associated to Ψ of the selected pattern. These two techniques share a type combination method that refines the existing type definitions by combining different definitions of the same type. For example, suppose pattern p is selected to write a formal expression and type t is initially declared as definition def_1 for specifying element e_1 of p. When specifying element e_2, *property-guided declaration* leads to a suggestion that t should be defined as definition def_2 to enable the correct representation of the value assigned to e_2. If def_1 is not equal to def_2, the combination method will be applied to refine def_1 with def_2 by combining them into a new definition for declaring t.

If the generated intermediate result contains informal expressions, formalization of the result is needed. Since it is performed by applying the patterns indicated by the informal expressions, *function-related declaration* can be repeatedly manipulated to further refine the data types of the specification. When the formalization process terminates with a formal expression, a refined data type environment is obtained. Finally, *expression update* is carried out where all the formal expressions that are inconsistent with the refined type information are updated.

Serving as the critical techniques in the described declaration approach, *function-related declaration* and *expression update* will be presented in details respectively.

5.2.2 Function-related declaration

Function-related declaration guides the refinement of data types to enable the application of the selected specification patterns, i.e., the formalization of the intended requirements. It adopts *property-guided declaration* and *priority-guided declaration* in declaring data types for specifying element and generation intermediate result, respectively. Before presenting the detailed techniques in *function-related declaration*, some necessary concepts are introduced first.

Constants and variables compose a data context under which formal expressions in formal specifications can be written and become analyzable. The formal definition of *data context* is given as follows.

Definition 5 *A data context is a 4-tuple (C, T, V, vt) where C is the set of constants, T is the set of custom data types, V is the set of variables and $vt : V \to T$ is the type function that determines the data type of each variable in V.*

To facilitate automated analysis and improve specification readability, each variable in the data context is required to be defined as a custom type in our approach, i.e., for each $v \in V$, there exists a type t in T that satisfies $vt(v) = t$. For example, when describing an ATM system, a password should be defined as a variable of a custom type declared as *string*. Although the built-in type *string* itself is capable of representing the nature of password and one can define the required password as a variable of *string* type, it fails to distinguish the object from others that are also defined as *string*, such as error messages. In addition, modification on the definitions of all the password entities in the specification can be easily manipulated by modifying the definition of the corresponding custom type.

Definition 6 *Given a data context dc and a pattern p,* $es_p^{dc} : E_p \rightarrow choice \cup Exp_{dc} \cup \mathcal{P}(Props_{dc}) \cup strValue \cup numValue \cup typeValue \cup \mathcal{P}(E_p)$ *is an* element state *of p under dc revealing the value of each element* $e \in E_p$ *where*

- Exp_{dc} *is the universal set of formal expressions within context dc and each* $exp_{dc} \in Exp_{dc}$ *is a sequence:* $N^+ \rightarrow C_{dc} \cup V_{fsc} \cup Operator$ *where Operator is the set of operators in formal notations*
- $Props_{dc}$ *denotes the universal set of property values within context dc and for each prop* $\in Props_{dc}$*,* $inVar(prop)$ *is adopted to denote the variables involved in prop.*

Each element state reflects the state of the application process of the corresponding pattern where all the elements of the pattern are assigned with a specific value. Since the value of elements invovles the use of type definitions of the formal specification to be constructed, the data context of the formal specification is involved in the definition of element state. It should be noted that es_p denotes all the possible element states of p, i.e., set $\{es_p^{dc_1}, ..., es_p^{dc_i}, ...\}$ where $\{dc_1, ..., dc_i, ...\}$ is the universal set of data contexts.

Definition 7 *Given a data context dc and a pattern p, function* $satisfy_p^{dc} : PR_p \times ES_p^{dc} \rightarrow boolean$ *denotes satisfaction relations between pattern constraints and element states where each* $es_p^{dc} \in ES_p^{dc}$ *is a possible element state of p under dc and* $satisfy_p^{dc}(pr, es_p^{dc})$ *indicates* $\forall_{e \in \Delta(pr)} \cdot es_p^{dc}(e) \neq \varnothing \wedge pr$ *is satisfied by* es_p^{dc}*.*

In the definition, function $\Delta : PR \rightarrow \mathcal{P}(E \cup \{p\})$ indicates the objects involved in each constraint $pr \in PR$ in the corresponding pattern p. The objects may include the pattern p itself and the elements in E item of p.

Definition 8 *Given a data context dc and a pattern p,* $condSatisfy_p^{dc} : es_p \rightarrow \mathcal{P}(\ominus_p)$ *is a* conditional satisfaction *function iff*

- $es_0 \in es_p \wedge \forall_{e \in dom(es_0)} \cdot es_0(e) = \varnothing \Rightarrow condSatisfy_p^{dc}(es_0) = \ominus_p$*(The initial data context satisfies all the constraints in the* $\ominus$ *item of* p*)*

- $condSatisfy_p^{dc}(es_p^{dc}) = R \Rightarrow$

  $\forall_{PR_i \in dom(R)} \cdot \forall_{pr \in PR_i} \cdot (satisfy_p^{dc}(pr, es_p^{dc}) \vee$

  $((\exists_{e \in \Delta(pr)} \cdot es_p^{dc}(e) = \varnothing) \wedge (pr, es') \notin dom(satisfy_p^{dc})))$

  where $es' \subset es_p^{dc} \wedge \forall_{e \in dom(es_p^{dc} - es')} \cdot es_p^{dc}(e) = \varnothing \wedge (\forall_{e' \in dom(es')} \cdot es(e') \neq \varnothing)$

The above two definitions formally specify the two kinds of relations between the constraints in the corresponding pattern and a given element state. In the first relation, each element in the element state has been assigned with a specific value and these values satisfy the given pattern constraints. In the second relation, there exist elements that have not been specified in the given element state. All the specified elements hold the values that satisfy the given constraints in the $\ominus$ item of p.

Based on the above definitions, we will start to describe the techniques in the type declaration approach. According to the outline of the approach, the type combination method is employed in both *property-guided declaration* and *priority-guided declaration*, it is thus first introduced.

5.2.3 Type combination

Type combination is an operation that combines two different definitions of the same type into an appropriate new definition for declaring that type. The result of the operation is determined by common properties held by the definition pair. Considering that it is impossible to combine all kinds of definition pairs automatically, the strategy of the operation is to deal with syntactic issues by machines and ask the developer to handle the semantic problems.

In order to precisely describe various properties of definition pairs, the concept of *subtype* is introduced and formally defined as follows.

Definition 9 *Given a custom type* ct*,* $subType(ct)$ *denotes the subtype of* ct *where*

- ct *is basic type* $\Rightarrow subType(ct) = \varnothing$

- ct *is composite type with each field* f_i *defined as type* $t_i \Rightarrow subType(ct) = \{(f_1, t_1), ..., (f_n, t_n)\}$

- *ct is product type with the ith field defined as type* $t_i \Rightarrow subType(ct) = \{1 \rightarrow t_i, ..., n \rightarrow t_n\}$
- *ct is set or sequence type with each element defined as type* $t \Rightarrow subType(ct) = t$
- *ct is mapping type with domain defined as type* t_i *and range defined as* $t_j \Rightarrow subType(ct) = (t_i, t_j)$

Based on the definition, we try to summarize possible properties of definition pairs and figure out the corresponding combination solutions. Table 5.3 (with formal notations in SOFL) shows part of the work where $buildIn(t)$ denotes the built-in type that type t belongs to and def indicates the result definition of the combination operation. For each pair of type definition d and d' where $d \neq d'$, a combination solution $sol(d, d')$ can be found by matching the definition pair with the properties listed in the table.

It can be seen from the table, properties of definition pair are classified into two categories: properties where d and d' belong to the same built-in type and properties where d and d' belong to different build-in types. The first category is further divided into five sub-categories that cover all the built-in types (in SOFL) and a solution is provided for each specific property within each built-in type. For example, the first "basic" denotes the property that d and d' belongs to the same basic type and its corresponding solution "human effort" indicates the combination of such kind of definition pair needs intelligent decision and the developer will be asked to give the operation result based on d and d'. More specific properties are provided with combination solutions if both d and d' are composite types. The second property within "composite" category and its corresponding solution mean that if d and d' owns the same fields and some of them are declared as different types, the combination method should be conducted on each pair of different types to achieve $sol(d, d')$. Within the second category, all combinations of different built-in types are considered and only parts of them are listed in the table for the sake of space. For instance, if d and d' are declared as different basic types, only human effort is able to figure out the proper definition. In case d is a composite type and d' is a set type, the combination result should be d if one of the fields in d is defined as d'.

For some kinds of definition pairs requiring for human effort in the table, there may exist automatic or semi-automatic methods to obtain the combination results. We will carry out more case studies on more large-scale software systems to explore these methods.

Table 5.3. Solution table for type combination

Property of definition pair			**Combination solution**
$buildIn(d) = buildIn(d')$	basic		human effort
	set/sequence		$sol(subType(d), subType(d'))$
	composite	$subType(d) \subset subType(d')$	$def = d'$
		$dom(subType(d)) = dom(subType(d')) \wedge \exists_{(f,t)\in subType(d),(f',t')\in subType(d')} \cdot f = f' \wedge t \neq t'$	$\forall_{(f,t)\in subType(d)} \cdot \exists_{(f,t')\in subType(d')} \cdot t \neq t' \Rightarrow sol(t, t')$
		......	......
	product	$rng(subType(d)) \subset rng(subType(d'))$	$def = d'$
		......	......
	map	$dom(d) = dom(d') \wedge rng(d) \neq rng(d')$	$sol(rng(d), rng(d'))$
		$dom(d) = rng(d') \wedge rng(d) = dom(d')$	human effort
		......	......
$buildIn(d) \neq buildIn(d')$	$buildIn(d)$ and $buildIn(d')$ are basic types		human effort
	$buildIn(d) = composite \wedge buildIn(d') = set$	$\exists_{(f,t)\in subType(d)} \cdot t = d'$	$def = d$
		......	......
	$buildIn(d) = composite \wedge buildIn(d') = mapping$	$d = dom(d') \vee d = rng(d')$	$def = d'$
		......	......
	......	......	......

5.2.4 Property-guided declaration

Figure 5.16 shows the main procedure of *property-guided declaration* for each selected pattern p within data context dc where cE denotes the element currently being specified, e_0 denotes the first element to be specified in p, AR denotes the set of activated rules and $inc(pr)$ denotes that the developer identifies the property pr as being inconsistent with the expected function.

The main idea of *property-guided declaration* is to guide the declaration of data types according to the obtained constraints during the application of the corresponding pattern. If the obtained constraints involve the definition of data types, then they will be provided as guidance. If the existing data types lead to the violation of the obtained constraints, the developer will need to modify these data types and formalize the function with the updated type information.

Specifically, the elements in the item E of the pattern p are specified according to the rules in Φ of p. To facilitate understanding, we use $\Phi_p : i \times (E_p \cup \mathcal{P}(PR_p)) \rightarrow E_p \cup \mathcal{P}(PR_p)$ to denote all the rules in both Φ_E and Φ_R of p where i denotes sequence for applying these rules. For example, $2 \times e \rightarrow e'$ means that the element rule $e \rightarrow e'$ is the second rule to be applied and $3 \times PR \rightarrow PR'$ indicates that the constraint rule $PR \rightarrow PR'$ is the third rule to be applied. Element rules refers to the rules in the Φ_E item of p and constraint rule refers to the rules in the rule repositories of p. Since there are usually more than one candidate rules to be selected when applying the rule repository. Rule set Φ_p contains rules that are attached with the same sequence number. When dealing with these rules, the premise of each rule will be analyzed and the satisfied one will be activated.

For each $i(0 < i \leq$the length of $\Phi_p)$, if i corresponds to a set of constraint rules $R \subset \Phi_p$, the rule $(i, SPR) \rightarrow SPR' \in R$ will be identified where all the constraints in SPR can be satisfied. Meanwhile, a set of new constraints SPR' will be obtained and added to AR. If i corresponds to an element rule, cE will be set as $\Phi_p(i, cE)$ which is the next element waiting to be specified. To assist the value assignment to cE, activated rules that lead to constraints of cE will be extracted from set AR and these constraints form a constraint set GS. After confirming that all the constraints in GS are consistent with the desired function, the developer needs to assign a value to cE based on GS. In case that certain constraints in GS violate the expected function, the activated rules that lead to these constraints form a set AR' and will be deleted from AR. Then the developer will be required to specify

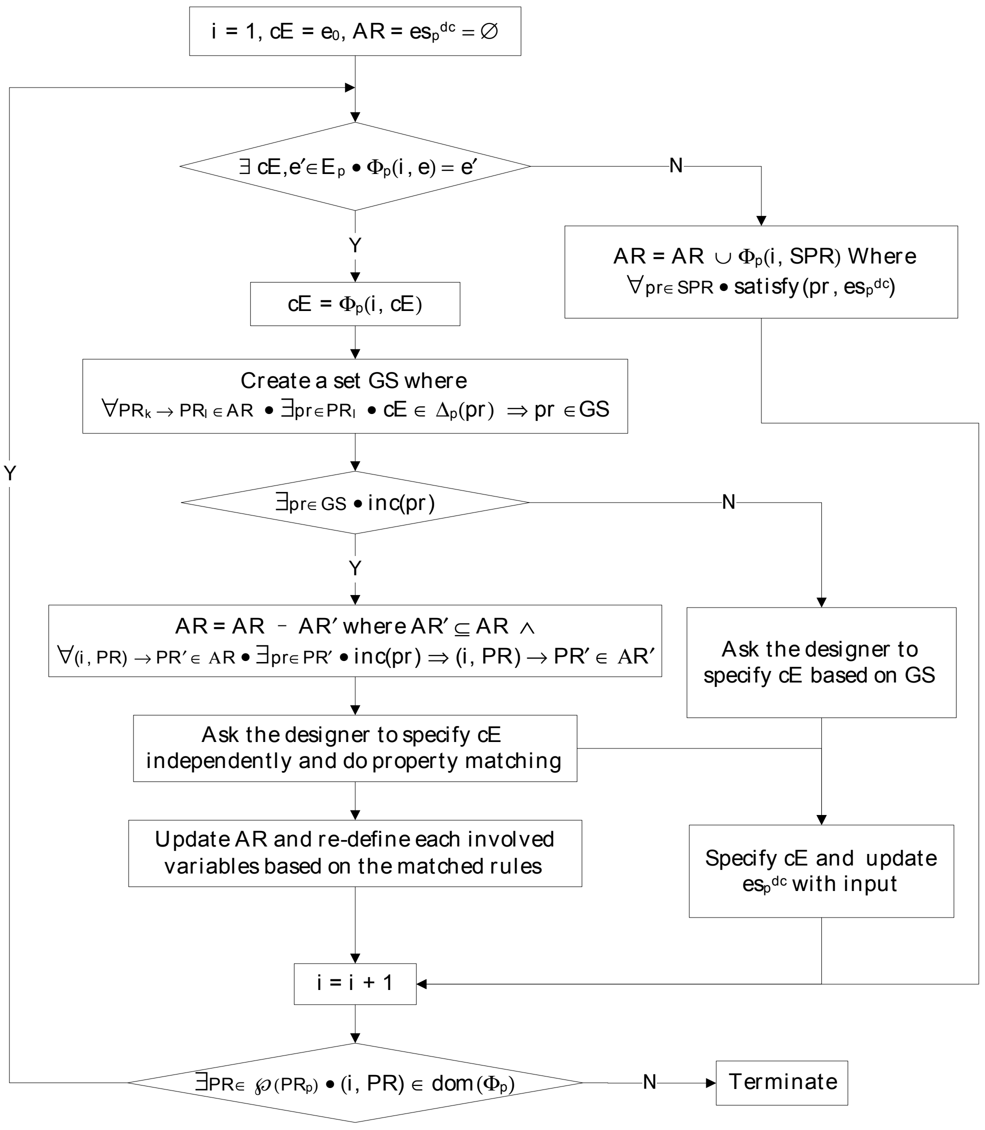

$i = 1$, $cE = e_0$, $AR = es_p^{dc} = \varnothing$
$\exists\, cE, e' \in E_p \bullet \Phi_p(i, e) = e'$
N
Y
$AR = AR \cup \Phi_p(i, SPR)$ Where $\forall pr \in SPR \bullet satisfy(pr, es_p^{dc})$
$cE = \Phi_p(i, cE)$
Create a set GS where $\forall PR_k \to PR_l \in AR \bullet \exists pr \in PR_l \bullet cE \in \Delta_p(pr) \Rightarrow pr \in GS$
Y
$\exists pr \in GS \bullet inc(pr)$
N
Y
$AR = AR - AR'$ where $AR' \subseteq AR \wedge \forall (i, PR) \to PR' \in AR \bullet \exists pr \in PR' \bullet inc(pr) \Rightarrow (i, PR) \to PR' \in AR'$
Ask the designer to specify cE based on GS
Ask the designer to specify cE independently and do property matching
Update AR and re-define each involved variables based on the matched rules
Specify cE and update es_p^{dc} with input
$i = i + 1$
$\exists PR \in \wp(PR_p) \bullet (i, PR) \in dom(\Phi_p)$
N
Terminate

Figure 5.16. The main procedure of the property-guided declaration

cE manually and property matching will be carried out to obtain the rules that match the given value.

In addition to the value v assigned to cE by the developer, set $CR : \mathcal{P}(\mathcal{P}(\Phi_p))$ serves as another critical participant in constraint matching which satisfies:

$$\forall_{R \in CR} \cdot (\forall_{(m,x),(n,y) \in dom(R)} \cdot m = n \wedge \forall_{R' \in CS-\{R\}} \cdot \forall_{(k,x') \in R,(l,y' \in R')} \cdot k \neq l)$$

$$\wedge \forall_{(no,Pr) \in dom(AR')} \cdot \exists_{R \in CR} \cdot \forall_{(no',Pr') \in dom(\Phi_p)} \cdot (no = no' \wedge \exists_{pr \in \Phi_p(no',Pr')} \cdot cE \in \Delta(pr))$$

Each set $R \in CR$ comprises all the candidate rules for substituting one of the rules that lead to constraints violating the expected function. With the given v, dc will be updated accordingly and property matching can be carried out by the following algorithm where RS denotes the set of rules that match the given v:

```
$RS = temp = \{\};$
$for\ each\ R \in CR\{$
  $for\ each\ Pr \rightarrow Pr' \in R$
     $if\ (satisfy_p^{dc}(Pr', es_p^{dc}) = true)$
        $temp = temp \cup \{Pr \rightarrow Pr'\};$
  $if\ (\mid temp \mid = 1)$
     $for\ the\ only\ element\ sr\ \ RS = RS \cup \{sr\};$
  $else\{$
     $tempP = \{\};$
     $for\ each\ Spr \rightarrow Spr' \in temp$
        $tempP = tempP \cup \{Spr\};$
     $display\ all\ the\ items\ in\ tempP\ and$
        $ask\ the\ developer\ to\ choose\ the\ most\ appropriate\ one\ "item";$
     $RS = RS \cup \{r\}\ where$
        $r \in temp \wedge \exists_{y \in \mathcal{P}(\Phi_p)} \cdot r = item \rightarrow y;$
     $\}$
$\}$
$return\ RS;$
```

This algorithm helps explore a set RS containing all the rules in $\mathcal{P}(\Phi_p)$ consistent with the function intended to be described which is reflected by the values assigned to elements. These rules will then be added into set AR and for each rule $Spr \rightarrow Spr'$, data context dc will be updated according to Spr.

5.2.5 Priority-guided declaration

The main idea of priority-guided declaration is to provide suggested definition of concerned types based on Ψ after assigning values to pattern elements. Rules in each Ψ are attached with priority attributes that help select a most appropriate one when elements are incompletely specified or no rule can be applied according to the specified elements.

Definition 10 *Given a pattern p,* $PS_p : \mathcal{P}(\mathcal{P}(\ominus_p))$ *is the* priority set *of p iff*

- $\forall_{ps_i \in PS_p} \cdot \exists_{es_p^{dc} \in es_p} \cdot condSatisfy_p^{dc}(es_p^{dc}) = ps_i$
- $\forall_{R \in \mathcal{P}(\ominus_p)} \cdot \exists_{es_p^{dc} \in es_p} \cdot condSatisfy_p^{dc}(es_p^{dc}) = R \Rightarrow R \in PS_p$

Definition 11 *Given a pattern p,* $\tau_p : \ominus_p \times PS_p \rightarrow N^+$ *determines the priority of each rule in* $\ominus_p$ *where* $\tau_p(r, ps_i) = n$ *means that* $r \in \ominus_p$ *is ranked as the nth rule in set* ps_i.

Based on the definition, priority-guided declaration is conducted as the following steps for each selected pattern p within formal specification context fsc.

1. Ask the developer to provide element information, and define types and variables when necessary, which results in an element state es_p^{fsc}.
2. Analyze priority set PS_p and extract the item $ps \in PS_p$ that satisfies $condSatisfy_p^{fsc}(es_p^{fsc}) = ps$.
3. Sort set ps into a sequence $psSeq$ where

   $\forall i, j : int \cdot 0 < i < j \leq \mid psSeq \mid \Rightarrow \tau_p(psSeq(i), ps) > \tau_p(psSeq(j), ps)$
4. Set $rule = psSeq(k)$ where k is initialized as 1. Provide the constraints involved in $rule$ for the developer to assist the declaration of relative types and variables.
5. If the suggestion is not accepted and $k \leq \mid psSeq \mid$, set $k = k + 1$ and repeat step 4-5. Otherwise terminate.

5.2.6 Expression update

In contrast to the traditional formal specification construction method that requires formal expressions to be written manually, function patterns enables automatic generation of formal expressions based on the given values of necessary elements. Therefore, instead of grammar checking, the essential idea of expression update in our approach is to record the element values specified during the pattern application process and reuse that information to update the original formal expression. For an expression *exp* generated through the application process *ap* of the pattern p, if *exp* becomes erroneous under the refined data context, it will be replaced by a new expression generated by applying p again based on *ap*.

Definition 12 *Given a pattern* p_0*, sequence* $(p_0, es^{p_0}_{dc_0}, exp_0, p_1, es^{p_1}_{dc_1}, exp_1, ..., p_n, es^{p_n}_{dc_n}, exp_n)$ *is the application process of* p_0 *where*

- $p_1, ..., p_n$ *are the reused patterns*
- *each* $es^{p_i}_{dc_i}$ *denotes the element state after all the elements in* E_{p_i} *are specified*
- exp_0 *denotes the intermediate formal result produced by applying* p_0 *with specified elements in* $es^{p_0}_{dc_0}$*, which can be represented as* $exp_0 = p_0(es^{p_0}_{dc_0})$
- *each* $exp_i (0 < i \leq n)$ *denotes the intermediate formal result generated by replacing certain informal part in* exp_{i-1} *with* $p_i(es^{p_i}_{fsc_i})$*, which can be represented as* $exp_i = exp_{i-1} \oplus p_i(es^{p_i}_{dc_i})$ *where* $exp_i = p_i(es^{p_i}_{dc_i})$ *if* $exp_{i-1} = \varnothing$
- exp_n *is the resultant formal expression*

Definition 13 *Given a data context dc,* $vdept_{dc} : T_{dc} \to V_{dc}$ *reveals dependent relations between types and variables where* $vdept_{dc}(t) = V$ *indicates that for each variable* $v \in V$*, the definition of* $vt_{dc}(v)$ *involves type* t*.*

Definition 14 *Given a data context dc and a pattern p,* $sdept^{p}_{dc} : T_{fsc} \to \mathcal{P}(es^{dc}_{p})$ *reveals dependent relations between types and element values where*

$$sdept^{p}_{dc}(t) = Es^{dc}_{p} \Rightarrow$$

$$\forall_{e \to vl \in Es^{fsc}_{p}} \cdot (vl \in Exp^{dc}_{p} \wedge \exists_{i \in N^+, v \in V_{dc}} \cdot (i, v) \in vl \wedge v \in vdept_{dc}(t)) \vee (vl \in Props_{dc} \wedge \exists_{v \in inVar(vl)} \cdot v \in vdept_{dc}(t))$$

Assume that the data context dc has been modified into dc', the update of each formal expressions exp previously written through application process $ap = (p_0, es^{p_0}_{dc_0}, exp_0, p_1, es^{p_1}_{dc_1}, exp_1, ..., p_n, es^{p_n}_{dc_n}, exp_n)$ is conducted as the following algorithm where $def_{dc}(t)$ denotes the definition of type t under dc.

```
$if(\exists_{(i,v)\in exp} \cdot (vt_{dc}(v) \neq vt_{dc'}(v)) \vee (\exists_{t\in T_{dc}} \cdot t \in T_{dc'} \wedge v \in vdept_{dc}(t) \wedge v \in vdept_{dc'}(t) \wedge def_{dc}(t) \neq def_{dc'}(t))$
{
  $exp_{-1} = \varnothing;$
  for each $p_i$ in $ap${
    $if(\exists_{t\in T_{dc}, e\rightarrow vl \in es^{p_i}_{dc}} \cdot t \in T_{dc'} \wedge$
      $def_{dc}(t) \neq def_{dc'}(t) \wedge e \rightarrow vl \in vdept_{dc}(t))$
        $exp'_i = exp'_{i-1} \oplus p_i(es^{p_i}_{dc'});$
    $else$
      $exp'_i = exp'_{i-1} \oplus temp$ where $exp_i = exp_{i-1} \oplus temp$
  }
  $exp = exp'_n;$
}
```

The algorithm first checks whether there exist variables or types used in exp with definitions being modified. If so, the application of pattern p_0 will be restarted with element information $es^{p_0}_{dc}$ and further formalization will be conducted by applying the rest of the reused patterns in ap with their element information sequentially. Before generating formal expression for each pattern p_i, the value of each element indicated by $es^{p_i}_{dc}$ will be analyzed to determine its change caused by the update of the data context. Expression exp_i can be directly used to formalize the current formal result exp'_{i-1} if no difference is found between $es^{p_i}_{dc}$ and $es^{p_i}_{dc'}$. Otherwise, $p_i(es^{p_i}_{dc'})$ will be produced to replace the corresponding informal part of exp'_{i-1}.

5.2.7 Case study

The processes *Account_confirm* and *Withdraw* in the CDFD of the banking system are used as examples to illustrate the data type declaration approach. For the process *Account_confirm*, manual declaration is first required for defining its inputs and outputs. According to the expected behavior of the process, one can easily respond with the following definitions:

$CustomerInf = composed\ of$

$\quad account\_num : string$

$\quad account\_psd : string$

$\quad end$

$inputInf, inf : CustomerInf, warning : string$

No pre-condition is needed in the process and the informal idea of the post-condition is that if the provided ID information can be found in the datastore *account_store*, data flow *inputInf* will be produced. Otherwise, error message *warning* will be displayed to the customer. Such idea leads us to the selection of the pattern *belongTo* as shown in Table 5.4 (The *expl* item is omitted for simplicity, and $\Phi = \varnothing$ means that no clarification rule is included in the pattern and the user needs to specify the two elements without guidance). This pattern is used to describe a relation where one object is part of another. There are three elements in the pattern: *element* denoting the member object, *container* denoting object that *element* belongs to and $specifier$ denoting the part of *container* that *element* belongs to. If element is specifier is assigned as null, constraints on their relations which can be assigned with either *null* or a constraint value. The application of pattern *belongTo* starts from the requirement of specifying these three elements. Apparently, *element* is inf and container is *account_store* which has not been defined. In case that $specifier$ is not decided yet, the generation of an intermediate result begins and *priority-guided declaration* will be carried out according to the priority knowledge given in Table 5.5. Suppose the developer uses "*AccountFile*" to represent its type, priority set $ps_1(\cup)$ is then selected and rule a is first suggested which indicates that the type *AccountFile* should be defined as $set\ of\ CustomerInf$. Assume that the suggestion is accepted, the formal expression for describing the "belongTo" relation is automatically generated and the post-condition of the process $Account\_confirm$ will be written as:

$if\ (inf\ inset\ account\_store)$

$then\ inf = inputInf$

$else\ warning = "Invalid\ user."$

Table 5.4. Pattern "belongTo"

f	belongTo
E	$\{element, container : expValue, specifier : null \mid PV\}$
PR	$\{dataType(element) = T, dataType(container) = set\ of\ T, specifier = f_i, specifier = null, ...\}$
Φ	$\varnothing$
Ψ	$\{\{dataType(element) = T, dataType(container) = set\ of\ T\} \xrightarrow{a} element\ inset\ container,$ $\{dataType(element) = T, dataType(container) = seq\ of\ T\} \xrightarrow{b} element\ inset\ elems(container),$ $\{dataType(element) = set\ of\ T, dataType(container) = T \to T'\}$ $\xrightarrow{c} belongTo(elemetn,\ dom(container)),$ $\{dataType(element) = set\ of\ T, dataType(container) = composite, specifier = f_i\}$ $\xrightarrow{d} element\ inset\ container.f_i,$ $\{dataType(element) = T \to T', dataType(container) = T \to T', specifier = null\}$ $\xrightarrow{e} element\ subset\ container,$ $\{dataType(element) = seq\ of\ T, dataType(container) = set\ of\ product, specifier = null\}$ $\xrightarrow{f}\ exists[e : container] \mid forall[i : N^+] \mid element(i) = e(i), ...\}$

Notice that no formal expression was written before the application of pattern *belongTo*, expression update is therefore not needed.

Since the data types and functions involved in the process *Selection* are simple enough to be manually written and the data context will not be affected after the description, data type declaration during the construction of process *Withdraw* is presented based on the type definitions declared for the process *Account_confirm*. Process *Withdraw* takes the intended currency type and amount as inputs and currency or error messages as outputs, which can be manually defined as:

$CurrencyType = string, Amount = real,$

Table 5.5. Priority in Pattern "belongTo"

Rule	**Priority set**				
	$ps_1(\cup)$	$ps_2(a,b,c,d)$	$ps_3(a,b,e)$	$ps_4(a,b,f)$	...
a	1	3	2	2	
b	2	4	3	3	
c	3	1	-	-	
d	4	2	-	-	
e	5	-	1	-	
f	6	-	-	1	
...					

$currencyType : CurrencyType, amount : Amount$

$currency : Amount, error : Msg$

The pre-condition is also true and the post-condition should clarify how the account information in *account_store* is altered when the withdraw operation is successfully done. Therefore, pattern *alter* will be selected to describe such function, which is shown in Figure 5.17 where $constraints(x)$ denotes certain constraints on object x.

It contains two elements for depicting the altering of system variables: *obj* denoting the object to be altered, *how* denoting the way to alter the specified object. If the whole given *obj* is replaced by a new value, element *how* is defined as a variable of the same data type as *obj*. If the requirement only modifies parts of the given *obj*, *how* will be defined as a set of items where each item specifies the operations performed on one kind of data items to be altered in *obj*.

During the application process of the pattern alter, property-guided declaration is carried out. Its detailed steps are given in Figure 5.18.

The above application process results in an definition "$CustomerInf \rightarrow AccountInf$"for type *AccountFile* that is more appropriate for describing process *Withdraw*. Thus, the original definition *set of CustomerInf*

f		alter
E		{*obj: expValue, how: nil*}
PR		{reuse, dataType(*obj)* = char, dataType(*how*) = dataType(*obj*), dataType(*obj*) = set, ∃ pair: *how* • (*how.pair.data.e.dom* = (dom = v_1) ∧ *how.pair.data*.e' = rng), ∃ pair: *how* • (dataType(*how.pair.oper)* = varValue), ...}
expl		alter → For describing the change of variables or parts of variables obj → The object to be altered how → The way to alter the given *obj* data → The data items to be altered oper → The operation performed to alter the data items how: set of composed of data, oper | varValue: varValue → "Choose your requirement from the following: 1. *obj* is altered by modifying parts of the included data items 2. *obj* is replaced by a new value." ……
Φ	**Φ_E**	{*obj* → *how*}
	Φ_R	*obj* → (R1, {r1}, γ) R1: {true} $\xrightarrow{r1}$ {*obj* : expValue} γ : r1 → ∅ *how* → (R2, {r2, r3, ⋯}, γ) R2: {dataType(*obj*) = char} $\xrightarrow{r2}$ {*how*: expValue} {dataType(*obj*) = map} $\xrightarrow{r3}$ {*how*: {set of composed of data, oper, expValue}} {dataType(*how*) != dataType(*objs*)} $\xrightarrow{r4}$ {re(*how*)} {*how*: set of composed of *data*, *oper*, dataType(*obj*) = map} $\xrightarrow{r5}$ {∃ pair∈ *how* • ((*pair.data*: {expValue, Composite}) (*pair.oper*: {varValue, Req, ⋯}))} {*how*: set of composed of *data*, *oper*, dataType(*obj*) = set of composed f1, ⋯, fn} $\xrightarrow{r6}$ {∃ pair∈ *how* • ((*pair.data*: {expValue, {f1, ⋯, fn}, constraint(f1), ⋯, constraint(fn)}) (*pair.oper*: {varValue, Req, ⋯}))} {dataType(*obj*) = map, *pair.data*: composite} $\xrightarrow{r7}$ {∃ pair∈ *how* • (*pair.data.e*: {constraint (dom), constraint(rng), constraint(dom, rng)} *pair.data.e'*: {dom, rng, both})} …… γ : r2 → {r4, ⋯}, r3 → {r4, r5, r6 ⋯}, r4 → {r2, r3, ⋯}, r5 →{r7, ⋯}, r6 → ∅, r7 → ∅, ⋯
Ψ		{!reuse, dataType(obj) = map, $\exists_1$ pair∈ how • ((pair.data.e = (dom = v)) (pair.data.e' = rng) pair.oper = p(e1, ⋯, en))} → "obj = override(~obj, p(e1, ⋯, en))" {!reuse, dataType(how) = dataType(obj)} → "obj = how" {!reuse, dataType(obj) = composed of f1, ⋯, fn, ∀ pair ∈ how • pair.data = fi pair.oper = p(e1, ⋯, en)} → "obj = modify(~obj," + ∀pair ∈ how • "pair.data → pair.oper)" + ")" {dataType(*obj*) = set of T, #*how* > 1, $\exists_1$pair ∈ *how* • *pair.data*: constraint(x) *pair.oper* = p(*x*)} → "let X = {xi: ~*obj* | constaint(xi)} in *obj* = union(diff(~*obj*, X), {" forall[xi: X] • "P(xi)""})" ……

Figure 5.17. The specification pattern *alter*

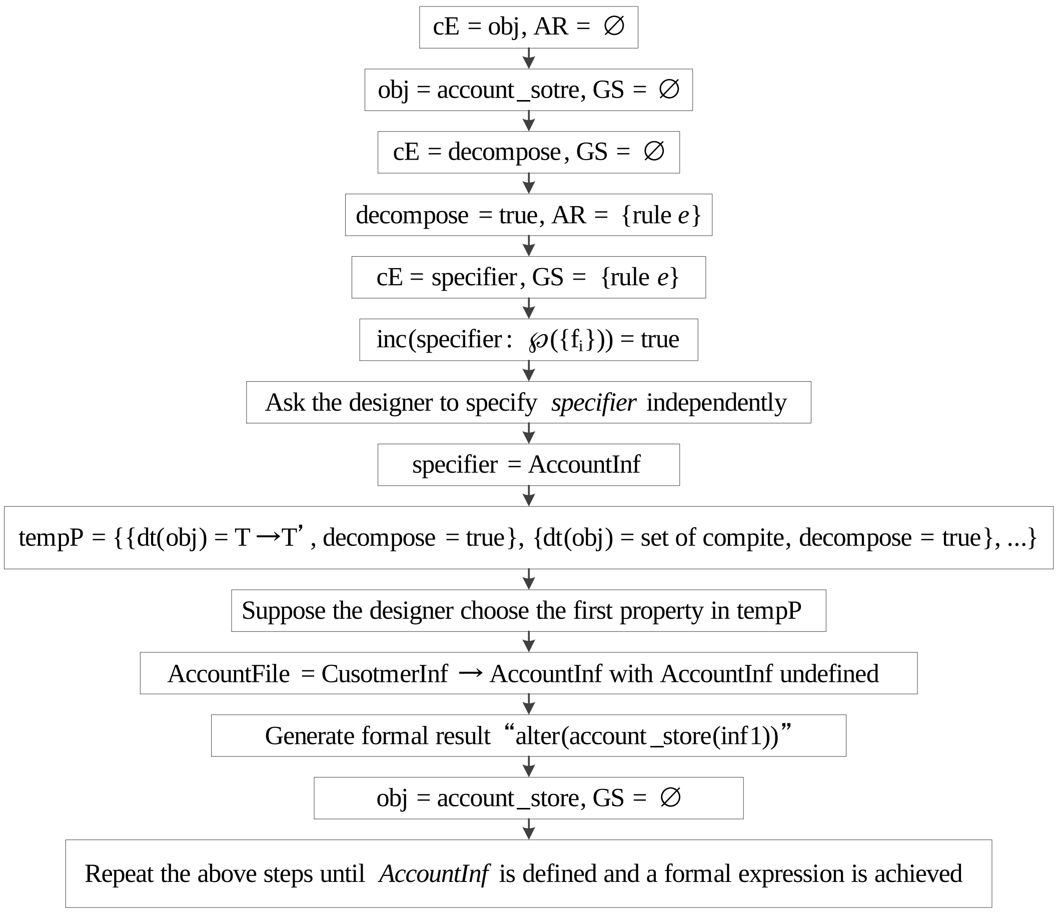

cE = obj, AR = $\emptyset$
obj = account_sotre, GS = $\emptyset$
cE = decompose, GS = $\emptyset$
decompose = true, AR = {rule *e*}
cE = specifier, GS = {rule *e*}
inc(specifier: $\wp(\{f_i\})$) = true
Ask the designer to specify *specifier* independently
specifier = AccountInf
tempP = {{dt(obj) = T →T', decompose = true}, {dt(obj) = set of compite, decompose = true}, ...}
Suppose the designer choose the first property in tempP
AccountFile = CusotmerInf → AccountInf with AccountInf undefined
Generate formal result "alter(account_store(inf1))"
obj = account_store, GS = $\emptyset$
Repeat the above steps until *AccountInf* is defined and a formal expression is achieved

Figure 5.18. The piority-guided declaration process during the application of the pattern "alter"

needs to be refined by applying the combination method. According to the solution table for type combination, the definition of type *AccountFile* should be refined as: $CustomerInf \rightarrow AccountInf$.

Due to the refinement of the definition of type *AccountFile* and the use of the type in the post-condition of process $Account\_confirm$, formal expression previously generated by applying the pattern *belongTo* needs to be updated accordingly. The application process of the pattern *belongTo* for the post-condition of process $Account\_confirm$ can be described as:

$(belongTo, \{element \rightarrow inf, container \rightarrow account\_store, specifier = null\}, "inf\ inset\ account\_store")$

According to the algorithm for expression update, formal expression "$inf\ inset\ account\_store$" will be transformed into:

$$belongTo(\{element \rightarrow inf, \quad container \rightarrow account\_store,$$
$$specifier = dom\}, newExp)$$

where $account\_store$ is defined as a map type and element $specifier$ is modified into "*dom*" in the refined data context. By analyzing the above expression in the context of the Ψ of the pattern *belongTo*, formal expression "$inf\ inset\ dom(account\_store)$" will be generated as the value of *newExp* to replace the original one.

5.3 Summary

In this chapter, we have presented how to guide requirements formalization by applying the specification pattern system. As we have mentioned, there are two activities interleavingly carried out during requirements formalization: describe software behaviors in formal expressions and data type declaration. These two activities are described in detail respectively and a case study is presented for illustrating each of them.

In the next chapter, we will propose a method for representing the pattern knowledge involved in the specification pattern system. Although the structures and formal definitions of the pattern and the pattern system are already given in the previous chapter, there will be some problems if we directly use them as the representation of the pattern knowledge. Our representation method is given to solve the problems. We will also give the utilization method for utilizing the pattern knowledge represented in the proposed representation.

Chapter 6

Representation of the pattern knowledge

Every kind of knowledge must be attached with a representation language to enable its availability to the users. Our pattern knowledge is no exception. Instead of directly using the structure and formal definition of the specification pattern and the pattern system to represent the pattern knowledge, we adopt two languages to represent different parts of the knowledge. The reason will be explained when presenting each of these two languages. One language is designed to represent the part of the knowledge that needs to be displayed to the developer and the other is designed to be applied by machines in an automated manner. Considering the different characteristics between the requirements of machine-oriented knowledge representation and human-oriented knowledge representation, attribute tree and HFSM are chosen to describe the two kinds of knowledge respectively.

It should be noted that the knowledge representation for data type declaration is not included in this chapter which will be studied in our future work.

6.1 Attribute tree

Tree structure is often adopted to describe hierarchical systems to facilitate the understanding of the system architecture. According to the definitions of elements' constituent types, the attributes of a complex requirement may form a hierarchy. For example, an element of *Req* type is described by a low-level requirement which is decomposed into low-level attributes. On the other hand, element definitions need to be shown to the developers to guide the clarification of the elements. If they are represented in formal notation or unstructured language, the produced guidance would be difficult to follow. Therefore, an attribute tree structure is designed to represent the element definitions. It visualizes the inner structures of each element definition and is meanwhile assigned with formal semantics. Through this representation, the developer can intuitively view the function details of

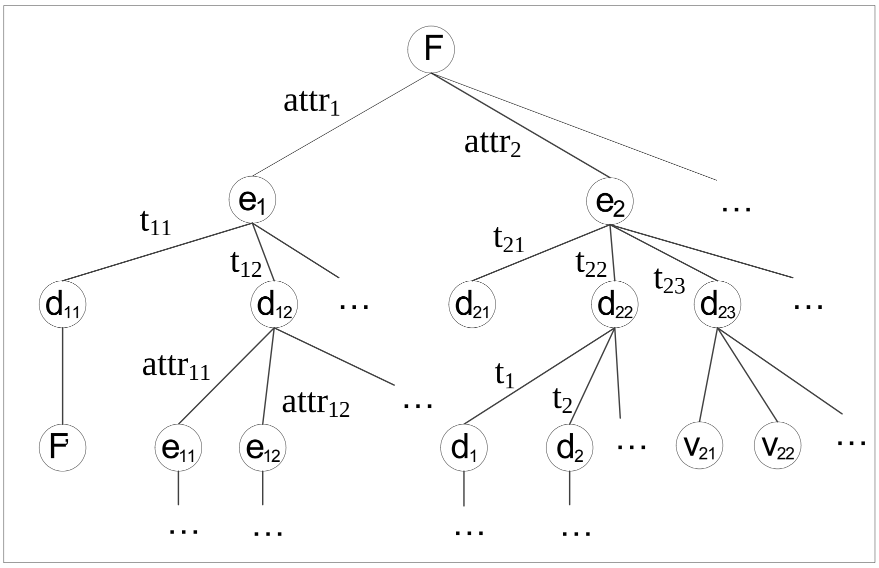


Figure 6.19. The structure of attribute tree

the requirements to be formalized and the clarified requirements can be automatically transformed into formal specifications.

6.1.1 The definition of attribute tree

Figure 6.19 shows the attribute tree representation.

The root node F of the tree denotes that the pattern is used to guide the clarification of the requirements on function F. Its child nodes $e_1, e_2, ...$ denote the requirement elements for composing the pattern. Each label $attr_i$ reveals that the element e_i is defined to represent attribute $attr_i$ of F. For each node e_i, its child nodes $d_{i1}, d_{i2}, ...$ indicate the def item of the corresponding element where each d_{ij} represents one of the constitute types.

For each node d_{ij}, label t_{ij} shows the type identifier of d_{ij} and the child nodes demonstrate the inner structure of d_{ij}. There are five kinds of inner structures for representing different constitute types.

- multiple child nodes denoting candidate items

  This structure is designed for representing *choice* type where each child node denotes a candidate item to be chosen. For example, the child nodes of node d_{23} indicate that a value of type t_{23} is given by selecting from $\{v_{21}, v_{22}, ...\}$.

- no child node

  Atomic types, except for *choice* type, are represented as leaf nodes, such as node d_{21}.

- one child node denoting another specification pattern

  Req types are represented by attaching a child node denoting the specification pattern for guiding the assignment of the corresponding element. For example, node d_{11} is attached with a child node F' where F' refers to a specification pattern represented as another tree structure. It indicates that if element e_1 is assigned with a value of constitute type t_{11}, it will be specified based on the specification pattern F'.

- multiple child nodes denoting child elements

  Structured types *Composite* and *Option* are represented by a set of child nodes. For *Composite* type, each child node denotes a child element of the corresponding element. For *Option* type, child nodes represent the pre-defined child element set to choose from. For example, node d_{12} is attached with a set of child nodes which can be further decomposed. If t_{12} is a *Composite* type, node d_{12} refers to the constitute type that decomposes element e_1 into child elements $e_{11}, e_{12}, ..., e_{1n}$. Otherwise, t_{12} is an *Option* type, which indicates that a value of t_{12} type should be given by specifying at least one of the child elements in $\{e_{11}, e_{12}, ...\}$.

- multiple child nodes denoting constitute types

  An element of *Set* type comprises a set of child elements defined with the same def item. Accordingly, *Set* type is represented by a set of child nodes, each of which denotes one of the constitute types involved in the shared def item. For instance, constitute type d_{22} decomposes element e_2 with a set of child elements. These child elements are defined with a same def item d where $d = d_1 \cup d_2 \cup ....$

The pattern *alter* is used as an example to illustrate attribute tree. The complexity of the alter operations on complicated data structures underlines the merits of using attribute tree to represent element definitions. The formal definition of the pattern *alter* is previously given in Figure 5.17. As we have presented, this pattern is used to the formalization of the functions that change variables or parts of variables (written in the first mapping of the *expl* item). It involves two elements *obj* and *how* representing the object to be altered and the way to alter the given object respectively. Element *obj* is initially defined as $expValue$ type and the definition of the element *how* cannot be determined at the beginning of the requirements formalization. Rules in item Φ guides the process of determining the definitions and values of these two elements. With the determined values, formal specifications can be generated according to the Ψ item.

This definition involves large amount of formal expressions for representing the element definitions; we only list some of them in the figure as examples, such as $how : set\ of\ composed\ of\ data, oper$. Imagine we use natural language to represent this definition as guidance to clarify *how*. It will be overwhelming for the developers when they are guided to the lower and lower elements of the elements *data* and *oper*. By contrast, an attribute tree that represent these definitions leads to guidance much more comprehensible. Figure 6.20 shows this attribute tree for the pattern *alter*. The two elements of the pattern are represented by nodes *obj* and *how* and their semantic is demonstrated by the labels attached to the corresponding branches. For an alter operation, element *obj* denotes the attribute of "the object to be altered" and *how* denotes the attribute of "the way to alter the object". Their definitions are reflected by the corresponding subtrees respectively.

According to the subtree of node *obj*, the def item of element *obj* is defined as d_{obj} which is of $expValue$ type. It indicates that there is only one way to specify element *obj* for any alter operation, which is to assign *obj* with a formal expression representing certain system variable.

The subtree of node *how* shows several different ways to assign element *how* for different requirements. Due to the sake of space, we only give two of them to illustrate the example pattern. Constitute type d_1 and d_2 indicates two kinds of requirements on altering the given *obj*. The first kind is to alter part of the given *obj*, which is clarified by assigning a value of *Set* type d_1 to *how*. Each child element of the value specifies the operation performed for modifying one kind of data item in the given *obj* (the operation is denoted as attribute *operAttr* and the kind of data item is denoted as attribute *dataAttr*). All the child elements share the same def item denoted as node d_0. According to the child nodes of d_0, each child element is of *Composite* type consisting of two lower-level elements *data* and *oper* which refer to the attributes *dataAttr* and *operAttr* of the child element respectively. The second kind is to replace the whole *obj* with a new value of *expValue* type d_2.

In the definition of constitute type d_1, element *data* is defined with four constitute types indicating four different ways to specify *data*.

- If the data items to be altered can be described in defined variables, a value of d_{01} type will be assigned

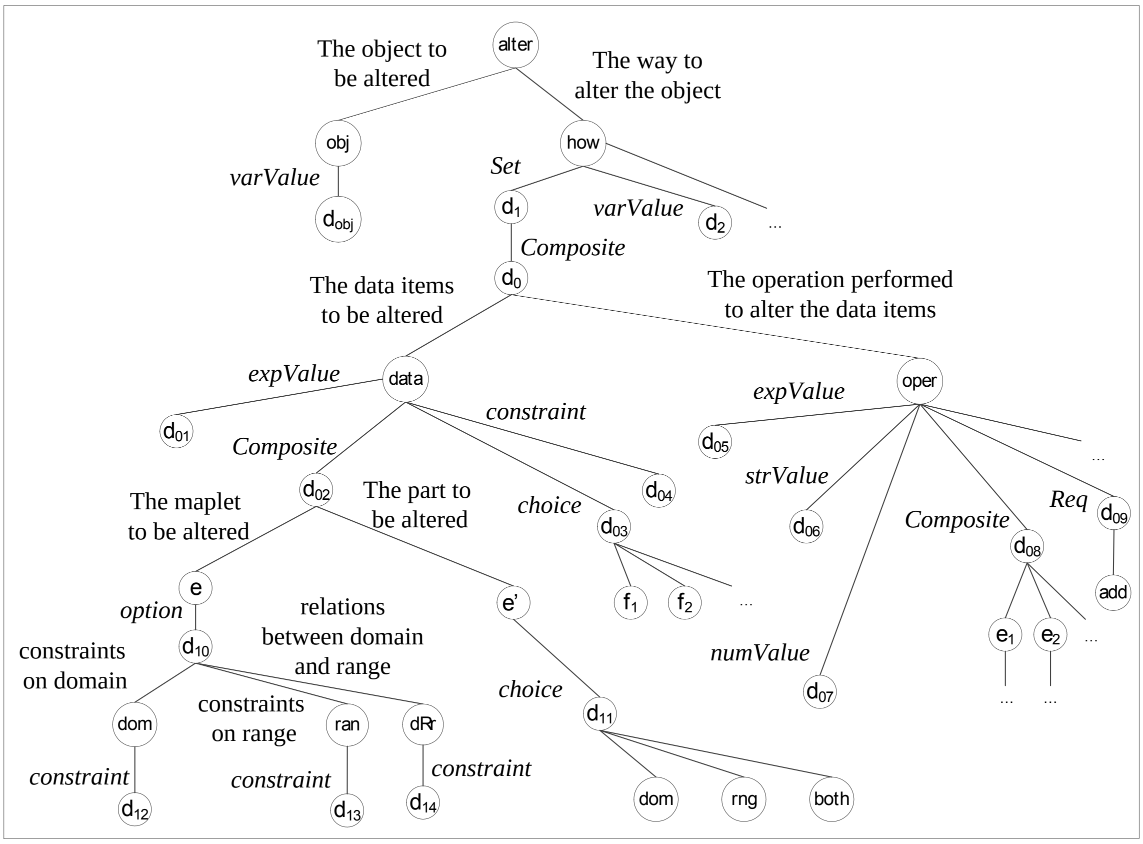


Figure 6.20. The attribute tree of the pattern *alter*

to *data*. For example, when describing the altering of a data item denoted as a defined variable v, *data* is specified as v.

- If the given *obj* is a mapping, a value of type d_{02} will be assigned to *data* to specify two attributes: what kind of maplets is intended to be altered (denoted as element e) and which part of the intended maplets need to be altered (denoted as element e'). Since the definition of e has already been presented when explaining the *Option* type, its detail is skipped here. For element e', a value of d_{11} type should be assigned by choosing from "domain of the intended maplet" (denoted as *dom*), "range of the intended maplet" (denoted as *rng*) and "both domain and range of the intended maplet" (denoted as *both*).
- If the given *obj* is a composite system variable with multiple fields $f_1, ..., f_n$, type d_{03} will be adopted which specifies element *data* by selecting the fields to be altered from $\{f_1, ..., f_n\}$.
- For *obj* of other data structures, such as set or sequence (built-in types in SOFL), a value of type d_{04}, i.e., a *constraint* value, will be assigned to element *data* to describe the data items to be altered.

The child nodes of element *oper* reveal three kinds of requirements on the operations performed on the data items denoted as element *data*. The first kind is to replace *data* with a value of basic type, such as node d_{05}, d_{06} and d_{07}. The second kind is to replace *data* with a composite value, such as node d_{08} which assigns a composite value $(e_1, e_2, ...)$ to *oper*. The third kind is to perform other operations on *data*. For example, node d_{09} specifies element *oper* with a low-level requirement on function *add*. It is adopted to describe the addition of new data items to *data*.

6.1.2 requirement tree

An attribute tree carries the information of elements definition. When all the elements are assigned with values of one of their constitute types, an attribute tree is transformed into a requirement tree that demonstrates the function details of the intended requirements. Figure 6.21 shows the definition of requirement tree.

The root node r indicates that the requirement tree represents the individual requirement r. There is a label R attached to r, which means that r is instantiated from specification pattern R and the whole structure of the tree is consistent with that of the tree representation of R. The child nodes of r indicate the elements for composing

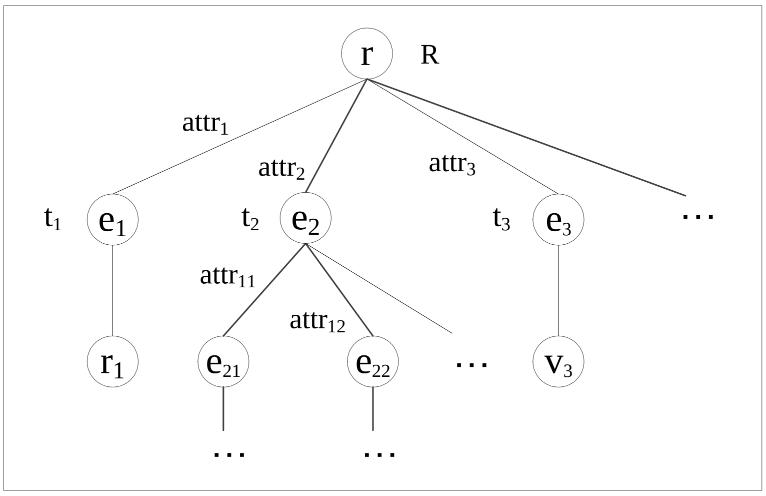


Figure 6.21. The strcuture of requirement tree

r and each attached label $attr_i$ denotes the attribute that element e_i refers to. Each e_i is decomposed into child nodes that reflect the value of the element e_i and the attached label t_i demonstrates the type of the assigned value. Corresponding to the five kinds of inner structures for representing element definitions, three kinds of structures are given to represent values of elements.

- one child node denoting an atomic value

  If an element e_i is assigned with an atomic value, its corresponding node will be decomposed into only one leaf node v_i which indicates the assigned atomic value. For example, the child node of node e_3 reveals that element e_3 is assigned with an atomic value v_3.

- one child node denoting a low-level requirement

  If an element e_i is assigned with a value of *Req* type, its corresponding node will be decomposed into one leaf node r_i which refers to another requirement tree representing the requirement for specifying e_i. For example, node e_1 is attached with child node r_1, which means that element e_1 is specified by requirement r_1.

- multiple child nodes

  Elements assigned with values of *Composite*, *Option* or *Set* type are represented by multiple child nodes where each child node denotes a child element. For example, element e_2 is decomposed into child elements $e_{21}, e_{22}, ...$, meaning that the value of e_2 comprises the values of its child elements.

Based on the above representations, clarifying requirements using specification patterns is to instantiate the corresponding attribute tree into requirement tree. Depth-first strategy is adopted in the construction process so that one can focus on the clarification of one attribute at one time (We first present the fundamental process of the construction of requirement tree without considering the use of other parts of knowledge in the pattern). Specific guidelines are given as follows.

- Preserve all the branches connecting the root node and its child nodes
- When reaching an element node, select from its child nodes according to the intended requirement and only traverse the subtree of the selected node
- When reaching a type node, create child nodes for the corresponding element node according to the intended requirement
- With the specification patterns, requirements clarification is to specify the relevant elements by assigning each element with a value of one of its . Such an activity is guided by the tree representation of the appropriate pattern and leads to a requirement tree

We will take the *withdraw* function of the ATM system as an example. For each customer (identified as $inf : CustomerInf$) intending to withdraw wa amount of currency of cy type, the ATM system dispenses the required currency and accordingly performs update operation on the balance and transaction information of the corresponding account in the data store *account_store*. Since the update operation is an alter operation, the requirement (denoted as r_1) for describing it can be instantiated from the specification pattern *alter*. Its clarification is guided by traversing the corresponding pattern tree in Figure 6.20, which results in the requirement tree shown in Figure 6.22 (Some labels for presenting the same kind of attributes are omitted for simplicity, such as the attribute label of child node *obj* of node r_2).

The traverse of the pattern *alter* starts from its root node and first reaches node *obj*. According to the given guidelines, the branch connecting nodes *alter* and *obj* is preserved. Since element node *obj* has only one child node d_{obj}, no selection decision needs to be made and element *obj* is required to be assigned with a value of d_{obj} type. In our case, data store *account_store* is the object to be altered; a value node v_1 is created as the child

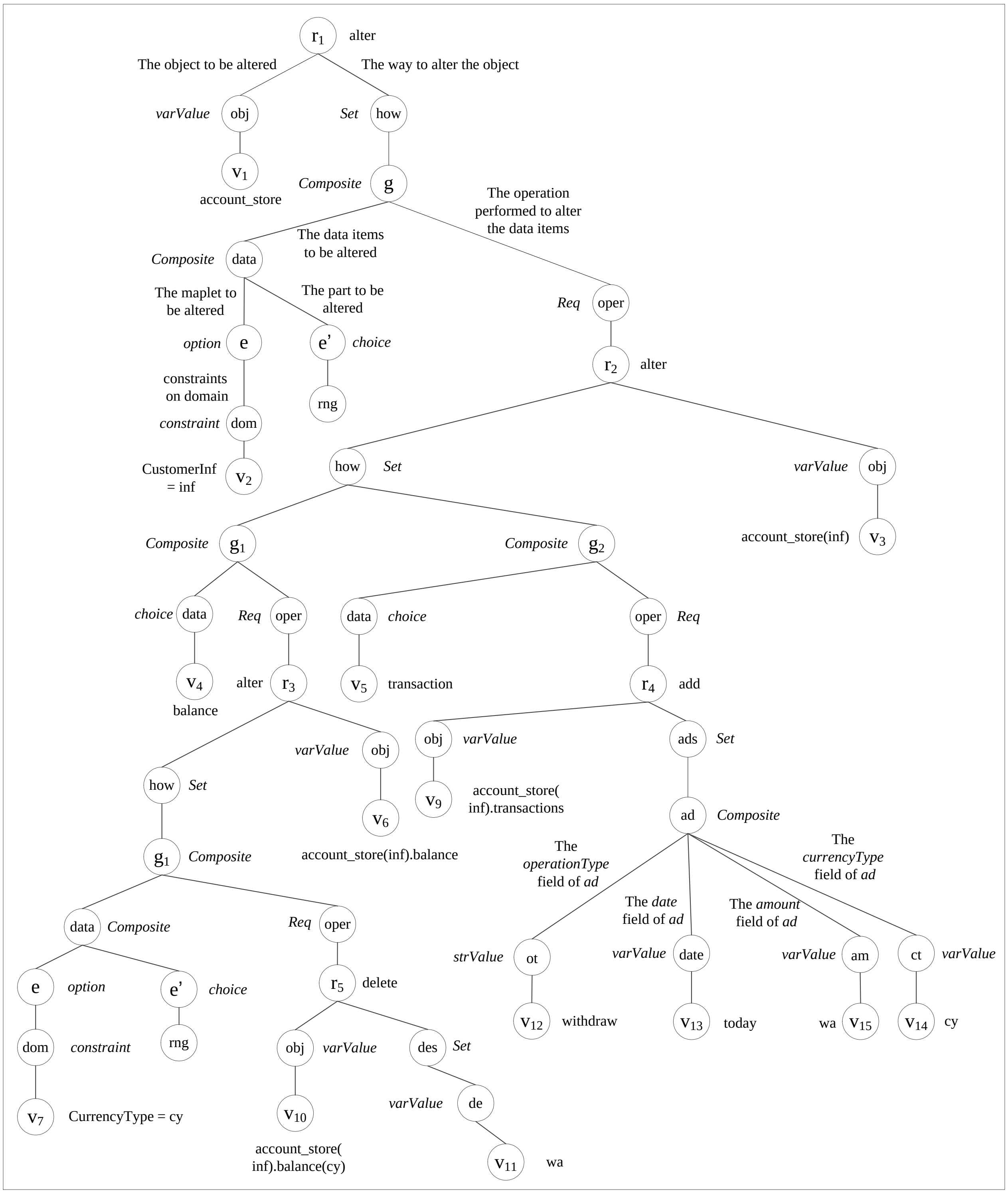


Figure 6.22. The requirement tree of the example alter operation

node of element node obj. Till now, the traverse of the left subtree has finished while the left subtree of the target requirement tree is constructed.

Requirements clarification of the example update operation continues with the right subtree of the pattern $alter$. When reaching element node how, child node d_1 is selected since r_1 only modifies the account information of the customer inf, rather than replacing the whole $account\_store$ with a new value. According to type d_1, child node g is created for element node how and decomposed into child element $data$ and $oper$ where $data$ specifies the data items to be altered in $account\_store$ and $oper$ indicates the way to alter these data items. For element $data$, constitute type d_{02} is employed since $account\_store$ is a mapping from customer information to account information. Child elements e and e' are assigned as constraint "$CustomerInf = inf$" and "rng" respectively, meaning to modify the range of the maplet whose domain is evaluated as inf, i.e., to modify the account information of the customer identified as inf. For element $oper$, constitute type d_{09} is adopted since the way to alter the intended account information needs to be described by another requirement r_2 on $alter$ function.

Low-level requirement r_2 is clarified by instantiating pattern $alter$. Node v_3 assigns formal expression $account\_store(inf)$ to obj which represents the account information of the customer identified as inf. According to the semantic of the update operation, both balance and transaction fields of the corresponding account need to be modified. Therefore, two child elements g_1 and g_2 are created for element how where g_1 describes the update of the balance information and g_2 represents the update of the transaction information.

For element g_1, low-level requirement r_3 is built for specifying the operations performed on balance information. The element obj of r_3 is assigned as formal expression $account\_store(inf).balance$ meaning the balance of the corresponding account. The assignment of element how of r_3 is similar to that of r_1. Since balance is defined as a mapping from currency type to amount, constitute type d_{02} is adopted where child elements e and e' are assigned as "$CurrencyType = cy$" and "rng" respectively. Such an assignment indicates to modify the balance information by updating the amount of currency of type cy. How to update the amount is specified by element $oper$. As a deletion operation, $oper$ is described by a lower-level requirement r_5 instantiating from the specification pattern $delete$ shown in Figure 6.23.

To represent a delete operation, two elements are needed to be specified: obj and des where obj denotes the object from which the data items are deleted and des denotes the data items to be deleted. Element obj has

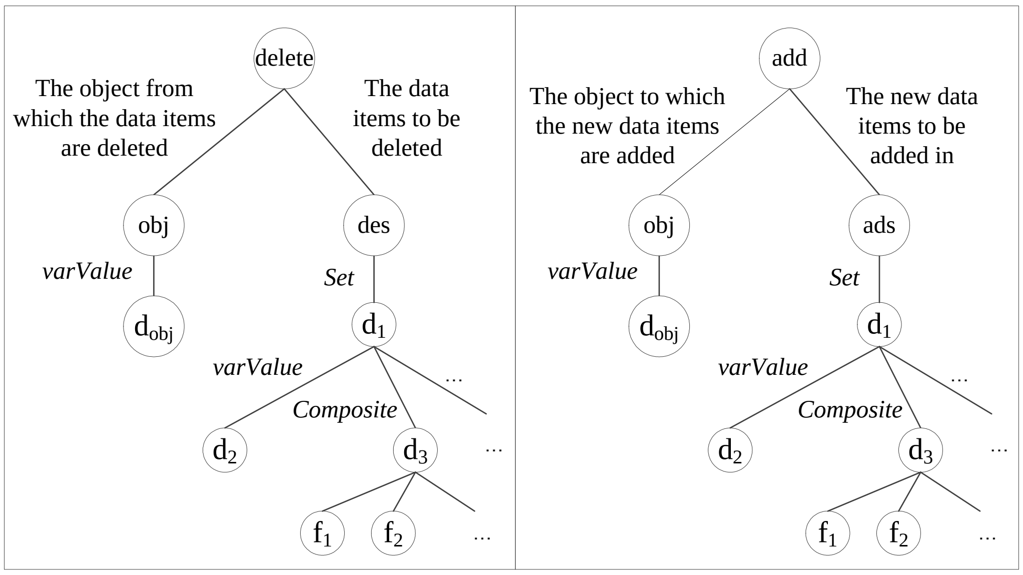

delete
The object from which the data items are deleted
The data items to be deleted
obj
des
varValue
Set
d_obj
d_1
varValue
Composite
d_2
d_3
f_1
f_2
add
The object to which the new data items are added
The new data items to be added in
obj
ads
varValue
Set
d_obj
d_1
varValue
Composite
d_2
d_3
f_1
f_2

Figure 6.23. The attribute trees of the pattern add **and** $delete$

only one constitute type d_{obj} and needs to be assigned with a value of $expValue$ type. Element des is defined as Set type where each child element represents one of the data items to be deleted. These child elements can be specified according to the constitute types $d_2, d_3, ...$ where d_2 is adopted to describe system variables to be deleted from obj and d_3 is used to describe composite values to be deleted. Based on the above definition for delete operation, element obj of r_5 is assigned with formal expression $account\_store(inf).balance(cy)$ representing the current amount of the currency of type cy. Element des of r_5 is attached with one child element de specified as withdraw amount wa, meaning to delete amount wa from the current balance of the currency of type cy.

For element g_2, a requirement r_4 instantiating from the specification pattern add is needed to describe the addition of the withdraw transaction. As a reference, Figure 6.23 shows part of the pattern add which is similar to the pattern $delete$. It defines two elements obj and ads to represent requirements for add operations where obj denotes the object to which the new data items are added and ads denotes the new data items. Element obj is of $expValue$ type while element ads is defined as Set type where each child element represents a new data item to be added in. Constitute types $d_2, d_3, ...$ are used to define these child elements. Based on the add pattern, r_4 is clarified by specifying elements obj and ads according to the semantic of the update operation. Since the transaction information is the object that the add operation performs on, its formal representation $account\_store(inf).transactions$ is assigned to element obj. For element ads, only one child element ad is attached which represents the withdraw transaction to be added in. Because each transaction consists of four fields: *date*, *operationType*, *currencyType* and *amount*, element *ad* is specified by four child elements corresponding to the four fields. According to the semantic of the example update operation, these four elements are assigned as *today*, *withdraw*, *cy* and *wa* respectively.

The example requirement is clarified in an intuitive manner since each element corresponds to an attribute of the requirement and the way to specify an element conforms to the nature of the corresponding attribute. Developers can easily build the requirement tree while analyzing the requirement. On the other hand, a better understanding of the function details can be obtained during the construction process.

6.1.3 Type tree

Supporting the formal description of system variables in formal expressions, the pattern *direct* in the retrieval category is designed with a distinguished attribute tree called type tree. We will explain the reason after the description of the tree. The main strategy is to utilize the type information included in the intended system variables and organize the information into a tree structure as the source of the formalization guidance.

Specifically, the type tree regards the data type of the intended system variable as its root node. Each tree branch $branch_i$ is represented as a transition (s, l, s') where s' is a child node of s, l is the label of the branch. Branches are divided into two kinds. If s' is a subtype of s, i.e., the definition of s relies on the definition of s'(denoted as $Dep(s, s')$), $branch_i$ is a downward branch where s' is a left child of s. If $Dep(s', s)$ establishes, $branch_i$ is a upward branch where s' is a right child of s. Downward branches use constraints on s' as their labels while upward branches take constraints on s as their labels. For example, suppose a function is to retrieve a system variable obj that satisfies two conditions: the data type of obj is $real * int$ and $obj(2) = 5$. It can be represented as a downward branch $(real * int, l_1, int)$ where l_1 is set to be "5" as the constraint on node int. In the case that the intended variable obj is an element of a set declared as $set\ of\ int$ that satisfies $obj > 5$, we should use a upward branch $(int, l_2, set\ of\ int)$ where l_2 is set to be "> 5" as the constraint on node int.

Nodes of the left subtree can only own downward branches while that of the right subtree are able to own both kinds of branches. And the rightmost leaf node of the right subtree corresponds to a defined variable serving as the basis of the target formal expression. Let's take the previous upward branch $(int, l_2, set\ of\ int)$ as an

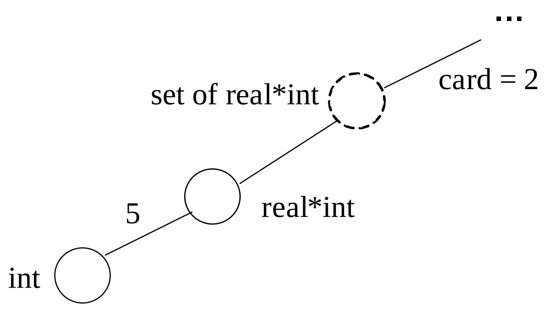


Figure 6.24. Pseudo node for special situations

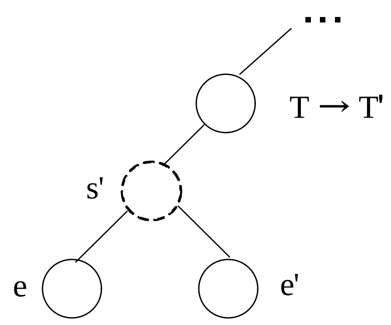


Figure 6.25. Pseudo node for special situations

example, assume that node *set of int* corresponds to a defined variable v, obj can then be presented as $(obj\ inset\ v) \wedge (obj > 5)$.

Besides, pseudo node is introduced to handle special situations. One situation is that certain node corresponds to more than one system variable. In the previous example, there might be a set of obj_i satisfying $obj_i(2) = 5$. Furthermore, the user may want to present a condition that the nodes of upper levels should satisfy by giving constraints on this set. For example, the intended variable becomes a sequence of product containing 2 obj_i where $obj_i(2) = 5$. To enable such kind of description in the tree structure, we create a pseudo node $set\ of\ real * int$ as shown in Figure 6.24. But if the root node happens to be in such situation, it will be turned into a pseudo node without creating a new one. The other situation is that some constraints have to be defined by composite values. For example, a node s identified as $T \rightarrow T'$ may be required to satisfy that one of the elements e in $dom(s)$ maps to the element e' in $rng(s)$. To express such meaning, a pseudo node $s\prime$ will be created as shown in Figure 6.25.

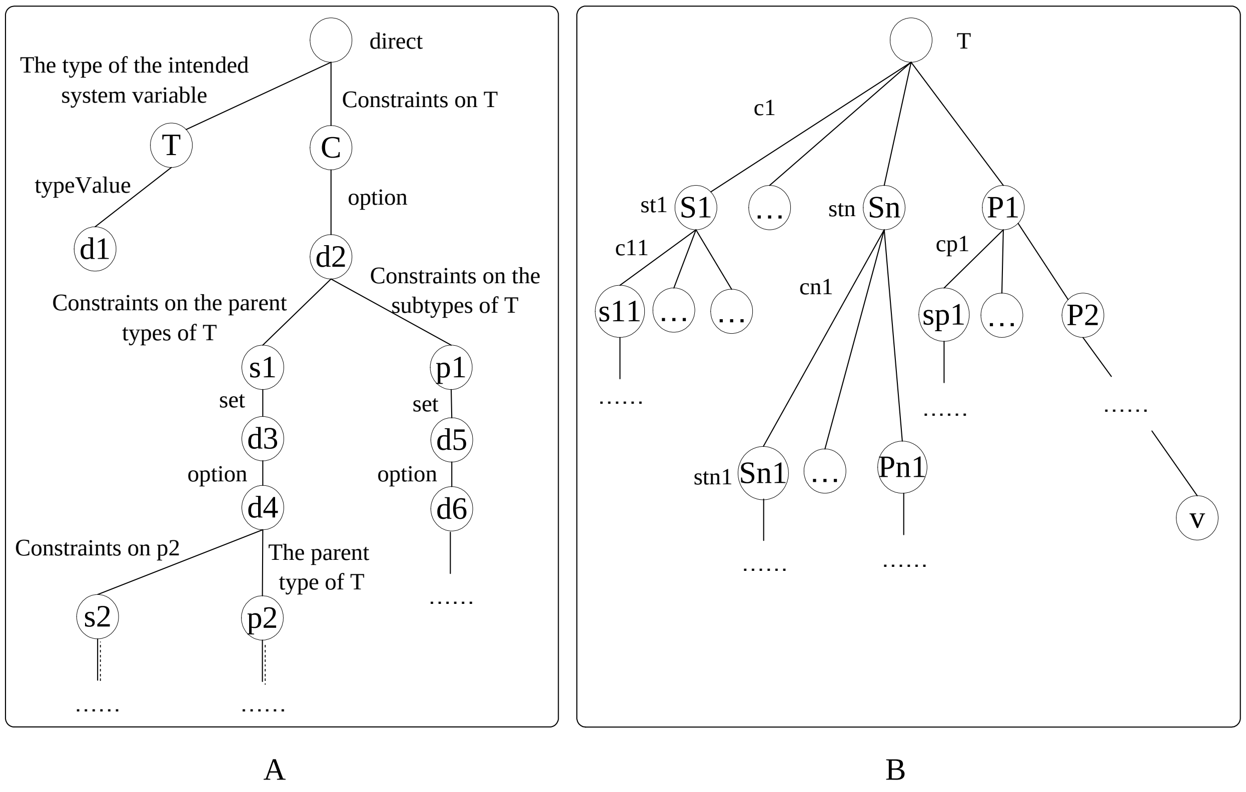


Figure 6.26. Comparison between attribute tree and type tree

Based on the above definition, we can compare the descriptions of system variables in two kinds of attribute trees. If we use the previously introduced attribute tree, the representation is given as shown in Figure 6.26A. If we use the specific type tree for the pattern *direct*, the representation is given as shown in Figure 6.26B where T denotes the type of the intended system variable..

As can be seen from the figure, although A is able to reflect parts of the structures of the intended system variable, it is not as intuitive as B. One can easily get the subtypes and parent types of each node by identifying its left and right child. And the labels on the branches clearly shows the constraints on the corresponding subtypes. Furthermore, the rightmost leaf node v demonstrate the defined variable that the intended system variable belongs to.

Based on the above concepts, the construction method of the tree can be given which only requires users' decisions on the semantic level and is therefore easy. During this process, developing branches is a critical operation that needs to be presented first. For a node s, its downward branches $branchD(s)$ are developed by setting selected subtypes of s as left child nodes and attaching given constraints on these subtypes as labels. By contrast, its upward branch $branchU(s)$ is only one new branch (s, l, t) where t is the selected type that takes s as its subtype, l is the given constraints on s. If s encounters special situations, pseudo nodes will be created following the instructions mentioned above.

The construction process starts from the root node s_0 which stands for the value of element $initialType$ meaning the data type of the intended variable. By generating downward branches for each node s that has been currently extended to, the left subtree will be built. In case that s_0 cannot develop downward branch, the left subtree will be empty. However, the right subtree can always be built by the following algorithm.

1. Generate $branchU(s_0)$ and set the current leaf node as the current node *cn*.
2. If *cn* is a right child node of the right subtree and there exists a defined variable v of type *cn* confirmed and accepted by the user, then quit with v.
3. If *cn* needs to be identified by constraints from its subtypes, then extend the left subtree of *cn* using the proposed method for left subtree generation.
4. If *cn* is not a left child of certain node of the right subtree, then generate $branchU(cn)$.
5. Set each current leaf nodes as the current node and repeat 2 – 5 respectively.

With a complete tree, the target formal expression *exp* is generated by applying the Ψ item of the pattern *direct*. We briefly introduce its inner mechanism which treats the left and right subtree of the root node separately. The expression *lExp* standing for the left subtree is obtained through the following algorithm, which can be skipped for the trees with empty left subtrees:

```
$create\ a\ stack\ currentNodes$;
$currentNodes.push(s_0)$;
$while(currentNodes\ is\ not\ empty)\{$
  $currentNode = currentNodes.pop()$;
$lExp = merge(Null, lExp, branchD(currentNode));$
 for each child $child_i$ of $currentNode$
   $\{currentNodes.push(child_i);\}\}$
```

And the algorithm to form the expression $rExp$ standing for the right subtree is as follows where $rLeaf$ denotes the leaf node of the right subtree:

```
$currentNode = rLeaf.parentNode;$
$rExp = v;$
$while(currentNode! = s_0)\{$
  $tempExp = Null;$
  $if(currentNode$ has left subtree$)\{$
      create a stack $temps;$
      $temps.push(currentNode);$
      $temp = currentNode;$
      $while(temps$ is not empty$)\{$
        $currentNode = temps.pop();$
        $tempExp = merge(Null, tempExp, branchD(currentNode));$
         for each child $tempChild_i$ of $currentNode$
           $\{temps.push(tempChild_i);\}\}\}$
  $rExp = merge(rExp, tempExp, \{branchU(temp)\});$
  $currentNode = temp.parentNode;\}\}$
```

Left subtrees are dealt with in a top-down manner while the right subtrees are transformed by a bottom-up method with the critical variable v as the start point. Finally after combining two expressions through the root node, the final formal expression is achieved as:

$exp = merge(rExp, lExp, \{branchU(s_0)\})$

Along the whole transformation process, function *merge* plays an important role which is defined as:

$merge : rExp * lExp * \ set\ of\ branch \rightarrow exp$

where *rExp* is a string denoting an expression transformed from a right subtree, *lExp* is a string denoting an expression transformed from a left subtree, and *exp* is a string denoting the expression generated by combining *rExp*, *lExp* and a set of branches. It is designed for constructing expressions under various situations, but the detailed rules are not further discussed for the sake of space.

Information display obtains data required by customers, which falls into the scope of the pattern *direct*. To better demonstrate the application of the pattern, we make the function more complicated on purpose. Consider describing displaying the July 3rd's transactions of accounts that have more than 3 transaction records on July 3rd, 500 US dollars and more than 1000 Japanese Yen. By applying method for constructing the type tree of the pattern *direct*, a tree structure shown in Figure 6.27 can be constructed.

Using the proposed transformation method, we will get the result formal expression as shown in Figure 6.28 where $output : set\ of\ Transaction$ denotes the output variable.

6.2 HFSM

6.2.1 The definition of HFSM

In addition to the knowledge that is visible to the developer, a larger part of the pattern knowledge is designed to be applied by machines. Despite the fact that pattern structure and formal definition provide a efficient way to organize the pattern knowledge, we treat the application process of the pattern system, rather than the pattern system itself, as knowledge and represent it using HFSM. The reasons are as follows. First, the application process specifies the guidance for each stage of requirements formalization, the method for utilizing such knowledge is straightforward without the need of analyzing the pattern system and its complexity stays independent from that

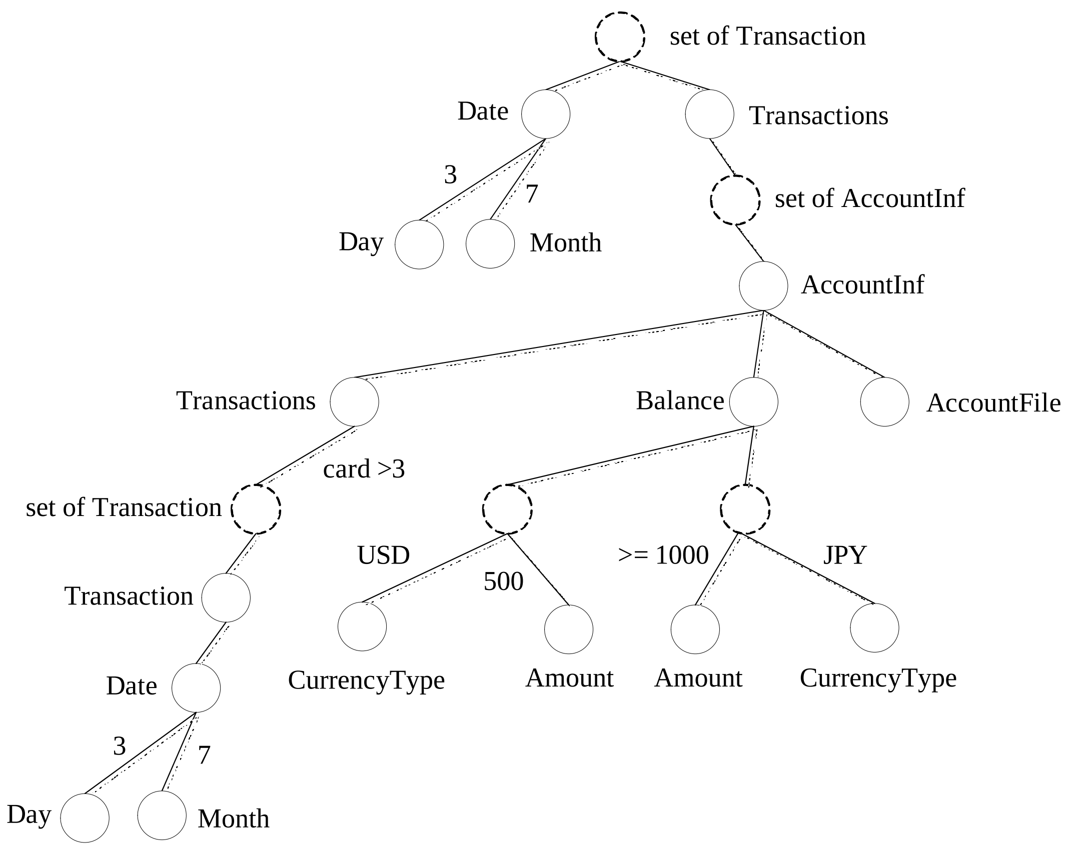


Figure 6.27. The type tree constructed for the example display function

```
let accountInfSet = {$item_{AccountInf}$ | $item_{AccountInf}$ : rng(account_store)
                            & let transactionSeq = [$item_{Transaction}$ |
                                  $item_{Transaction}$ : Transaction
                                  & exists[i : int] | $item_{AccountInf}$.transactions(i) = $item_{Transaction}$
                                             and $item_{Transaction}$.date(2) = 7 and $item_{Transaction}$.date(3) = 3]
                               in  len(transactionSeq) > 3
                              and $item_{AccountInf}$.balance(USD) = 500
                              and $item_{AccountInf}$.balance(JPY) >= 1000}
 in   let  TransactionsSet = {$item_{Transactions}$ | $item_{Transactions}$ :Transactions
                                   & exists[$item_{AccountInf}$ : accountInfSet] |
                                       $item_{AccountInf}$.transactions = $item_{Transactions}$}
      in   output = {$item_{Transaction}$ | $item_{Transaction}$ : Transaction
                            & exists[$item_{Transactions}$ : TransactionsSet, j: int] |
                               $item_{Transactions}$(j) = $item_{Transaction}$}
                             and $item_{Transaction}$.date(2) = 7 and $item_{Transaction}$.date(3) = 3
```

Figure 6.28. The formal expression generated based on the example type tree

of the pattern system. Second, as an interactive process, the application of the pattern system is described by the basic elements of interaction activities, such as input and output. Regardless of how the pattern system is modified, the definition of these elements will always remain the same. Therefore, the algorithm for utilizing the knowledge does not need to be modified even if the pattern system is updated. Third, HFSM can be used to accurately describe interaction processes and easily manipulated automatically. It allows the reuse of existing FSMs in describing more complex FSMs by introducing hierarchical relations among FSMs, which exactly matches the characteristic of the relations between individual patterns. Moreover, HFSM leads to a clear structure of the pattern knowledge and facilitates its maintenance. The definitions of FSM and HFSM are first given before explaining how to represent pattern knowledge.

Definition 15 *A FSM (Finite State Machine) is a 9-tuple* $(Q, q0, F, VP, I, G, \varphi, \delta, \lambda)$ *where* Q *is a non-empty finite set of states,* $q0 \in Q$ *is the initial state,* $F \subset Q$ *is the set of accept states,* VP *is a set of variable states where each variable state is a triple* (V, V', θ) *where* V *is the finite set of system variables,* V' *is a set of values and* $\theta : V \longrightarrow V'$ *defines the associated value for each* $v \in V$*,* I *is the finite set of symbols,* G *is the finite set of guard conditions,* $\varphi : Q \longrightarrow VP$ *is the state function indicating the values of the involved variables on each state,* $\delta : Q \times (I \times \mathcal{P}(G)) \longrightarrow Q$ *is the transition function relating two states by input and guard conditions,* $\lambda : Q \times (I \times \mathcal{P}(G)) \longrightarrow I$ *is the output function determining output based on the current state and input.*

In a FSM, each state denotes certain stage of the guidance production process, each $i \in I$ denotes a symbol for composing inputs and outputs, and each $g \in G$ denotes a constraint.

There are 2 kinds of FSMs: *value FSM* and *process FSM.* The former returns a value when terminated while the latter emphasizes on modeling an interactive process without returning any value. For each value FSM, state variable $return$ is created to carry the returned value.

Figure 6.29 shows an example FSM A where $Q_A = \{s_1, s_2, s_3, s_4\}$ (ΣC denotes that each $c \in C$ is provided as one of the choices for the developer, $\&c$ denotes the fact that item c has been selected and $req(var_2)$ denotes the request "specify system variable var_2").

Equations attached to states reflect the *state function* φ. When A is transferred to certain state s, system variables will be assigned according to the equations attached to s. For example, A will stay on initial state s_1

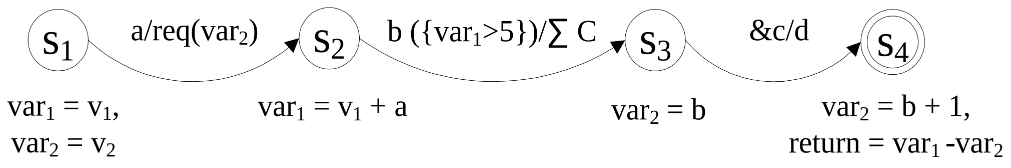


Figure 6.29. An example FSM model

when activated. The equations attached to s_1 indicate that system variable var_1 and var_2 will be initialized as v_1 and v_2 respectively. Notice that for each state s_i, if the value of certain variable v on s_i is the same as its value on the previous state of s_i, equations for assigning v will not be attached to s_i for simplicity. For example, no equation for assigning var_2 is attached to the state s_2, which means that the value of var_2 on s_2 is the same as the value on s_1. On the accept state s_4, "$return = var_1 - var_2$" is attached revealing that A is a value FSM that will return the value $var_1 - var_2$ when terminated.

Connecting states with arrowed lines, transitions reflect the *transition function* δ and *output function* λ of A. Each transition $s_i \rightarrow s_j$ is attached with a label $i(G)/o$ where $s_j = \delta(s_i, (i, G)) \wedge o = \lambda(s_i, (i, G))$, which means that when A stays on state s_i, if input i is received and each $g \in G$ is satisfied, output o will be displayed and A will be transferred to state s_j.

The semantic of the FSM A is explained as follows. Starting from the initial state s_1 where var_1 and var_2 are initialized as v_1 and v_2 respectively, A will be transferred to s_2 where var_1 is set as v_1+ a when receiving input "a". Meanwhile, an output that asks for the value of var_2 will be produced. If receiving response b while $var_1 > 5$ establishes, A will then be transferred to s_3 where var_2 is set as b and items in C will be provided for the developer to choose from. Finally, if $c \in C$ is selected, accept state s_4 will be reached where var_2 is set as $b + 1$ and A is terminated with output d and returned value $var_1 - var_2$.

Definition 16 *Given a FSM A and a state $s \in Q_A$, $Acc_A(s) \in \mathcal{P}(I \times \mathcal{P}(G))$ is an acceptable set on s iff $\forall_{acc \in Acc_A(s)} \cdot (\exists_{s' \in Q_A} \cdot \delta(s, acc) = s')$.*

For each state s in A, $(i, G) \in Acc_A(s)$ means that there exists a transition originated from s, which will be activated if input i is received and each $g \in G$ can be satisfied. Note that $(i, \varnothing) \in Acc_A(s)$ and $(\varepsilon, G) \in Acc_A(s)$ may also establish. The former indicates that a transition will be triggered on s when receiving i under any

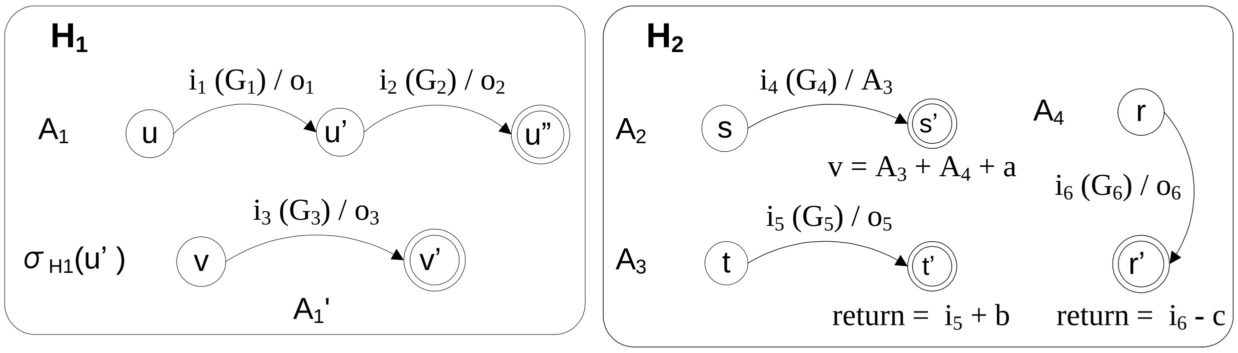


Figure 6.30. Example HFSM models

condition and the latter indicates that if each $g \in G$ is satisfied on s, a transition will be activated without any input.

Definition 17 *HFSM (Hierarchical FSM) is a pair (F, σ) where F is a set of FSMs and $\sigma : Q \cup I \cup V \longrightarrow \mathcal{P}(F)$ indicates the hierarchical relations among FSMs in F where lower-level FSMs interpret certain portion of upper-level FSMs iff $\exists_{A_0 \in F} \cdot \forall_{F' \in ran(\sigma)} \cdot A_0 \notin F'$ (A_0 is the root FSM).*

There are two kinds of hierarchical relations in σ: 1. lower-level FSMs demonstrate the inner transitions of states in upper-level FSMs; 2. upper-level FSMs utilize values generated by the FSMs in lower levels. In the second relation, a variable *return* is included in the system variables of each lower-level FSM for carrying the value to be used by the corresponding upper-level FSM. Figure 6.30 compares the two different relations through two example HFSMs H_1 and H_2. In H_1, FSM A_1 and the only FSM A_1' in $\sigma_{H_1}(u')$ hold the first relation where the detailed behavior of state u' is described by FSM A_1'. The second relation is held in H_2 where $\sigma_{H_2}(\lambda(s, (i_4, G_4))) = \{A_3\} \wedge \sigma_{H_2}(v) = \{A_3, A_4\}$. Label $i_4(G_4)/A_3$ indicates that if the corresponding transition is activated, the value generated by FSM A_3 will be displayed.

6.2.2 Representing the pattern knowledge in HFSM

Building a HFSM for representing the pattern knowledge starts from analyzing the application process of the pattern system (P, C, η) under the assumption that types and variables are already defined. As previously mentioned, for each function to be formally described, the first step is to help select a proper pattern from the pattern system. The second step is to apply the selected pattern, which can be further divided into two sub-steps: 1. guide the developer to specify the elements of the selected pattern according to the Φ item; 2. generate a formalization result by applying the Ψ item based on the specified elements and further formalize the result by applying the involved patterns. The explicit application process is given as follows where $p_{id=str}$ denotes the pattern with identity str, ce denotes the element currently being specified and apr denotes the set of all the satisfied constraints. To describe the HFSM representation of the pattern knowledge more clearly, we illustrate the application process of the pattern system more specifically.

Step 1 Pattern selection process

```
(a) $items = \eta(c_0)$
(b) Ask for choosing an item (denoted as $im$) from $items$
(c) $if(im \in C)\ then\ items = \{c_i : C \mid c_i \in \eta(im)\} \cup \{im_i : string \mid im_i = expl(eid)$ where $p_{id=eid} \in \eta(im)\}$
      and go to step b
    $else$   return $sid$ where $expl(sid) = im$
```

Step 2 Pattern application process (the application of $p_{id=sid}$)

```
(a) specifying elements
    $ce = e_0; apr = \varnothing;$
    $while$(elements of $p_{id=sid}$ are not completely specified){
      $if$($ce$ has not been specified){
        apply the rule repository of $ce$ and add the activated rules into $apr$;
        provide constraints on $ce$ in $apr$ as guidance and ask the developer to accordingly specify $ce$;
      }
      $ce = \Phi_E(ce);$
    }
(b) formal expression generation
    $foreach\ (PR_i \rightarrow str) \in \Psi\{$
```

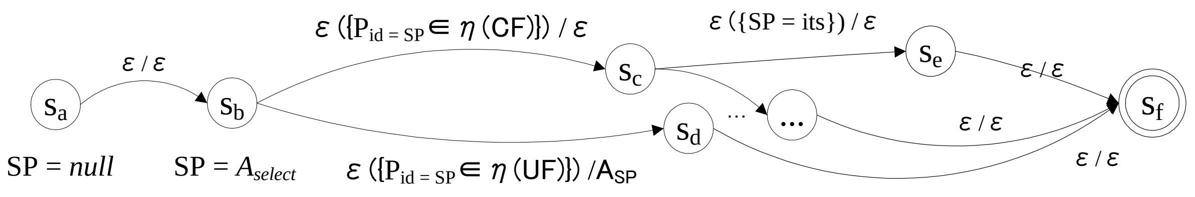


Figure 6.31. The top level FSM in the HFSM

```
$if(\forall_{pr \in PR_i} \cdot pr)$        $temp = \Psi(PR_i)$;
$if(temp$ is not a formal expression$)\{$
    $foreach$ informal statement $p(v_1, ..., v_n) \in temp\{$
        formalize it with element transmission mechanism;
    $\}$
$\}$
return $temp$;
```

According to the application process, the HFSM HF representing the pattern knowledge is built in a top-down way. The root FSM reflects the outline of the process by describing the initial and final states of steps 1 and 2. The details of the steps are modeled in lower-level FSMs. The lower-level FSM for modeling step 1 reveals the state transitions made by steps 1a, 1b and 1c. For step 2, sub-steps 2a and 2b is modeled by a set of lower-level FSM each describing the application of one of the patterns.

Figure 6.31 shows the root FSM A_{root} where SP denotes the system variable that holds the id of the selected pattern.

States s_a and s_b denote the initial state and final state of step 1 respectively. On s_a, SP is initialized as *null* indicating pattern selection has not been conducted. On s_b, SP is assigned as the value generated by traversing FSM A_{select} which models the details of step 1. For step 2, transition $s_b \rightarrow s_d$ denotes the application of UF patterns while paths between states s_c and s_f model the application of CF patterns. In transition $s_b \rightarrow s_d$, output A_{SP} indicates the formal expression generated by traversing FSM A_{sp} which models the details of the application process of pattern $p_{id=sp} \in \eta(UF)$. Originating from s_c, different transitions lead to different destination states where inner state transitions, i.e., the details of the application process of CF patterns, are described by lower-level

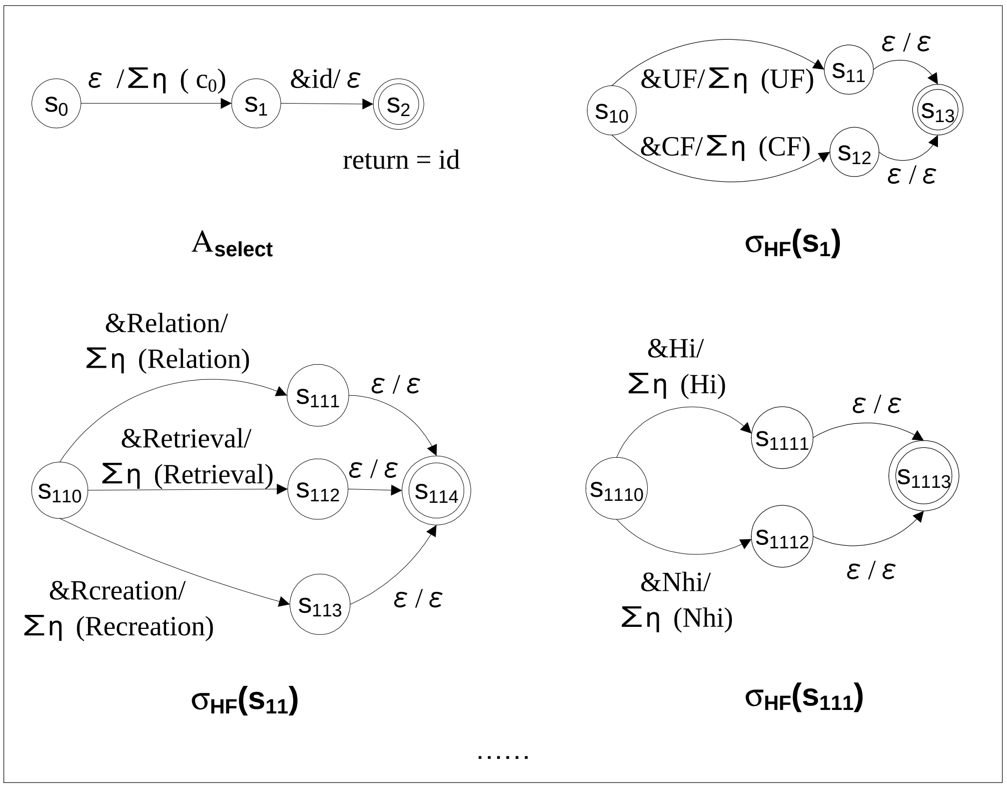


Figure 6.32. The HFSM describing the pattern selection process

FSMs. For example, transition $s_c \rightarrow s_e$ represents the application of the CF pattern ite where the inner transitions of state s_e, which indicate the details of the application process of the pattern ite, are left to be described by a lower-level FSM.

According to the above design, FSM A_{select} and a FSM set AS (where each FSM $A_{id_i} \in AS$ describes the detailed application process of an individual pattern with identity id_i) need to be built to interpret the detailed behavior of A_{root}.

FSM A_{select} is shown in Figure 6.32. It only provides the initial and final transitions of the pattern selection process. The former indicates that the top level categories of the pattern system will be provided for the developer to choose from at the beginning while the latter means that the developer will be guided to finally reach a proper pattern. Other details are given in lower-level FSMs organized in a hierarchy where each FSM interprets the inner state transitions of the corresponding upper-level state. For example, the FSM in $\sigma_{HF}(s_1)$ describes the inner state transitions of state s_1 in A_{select}.

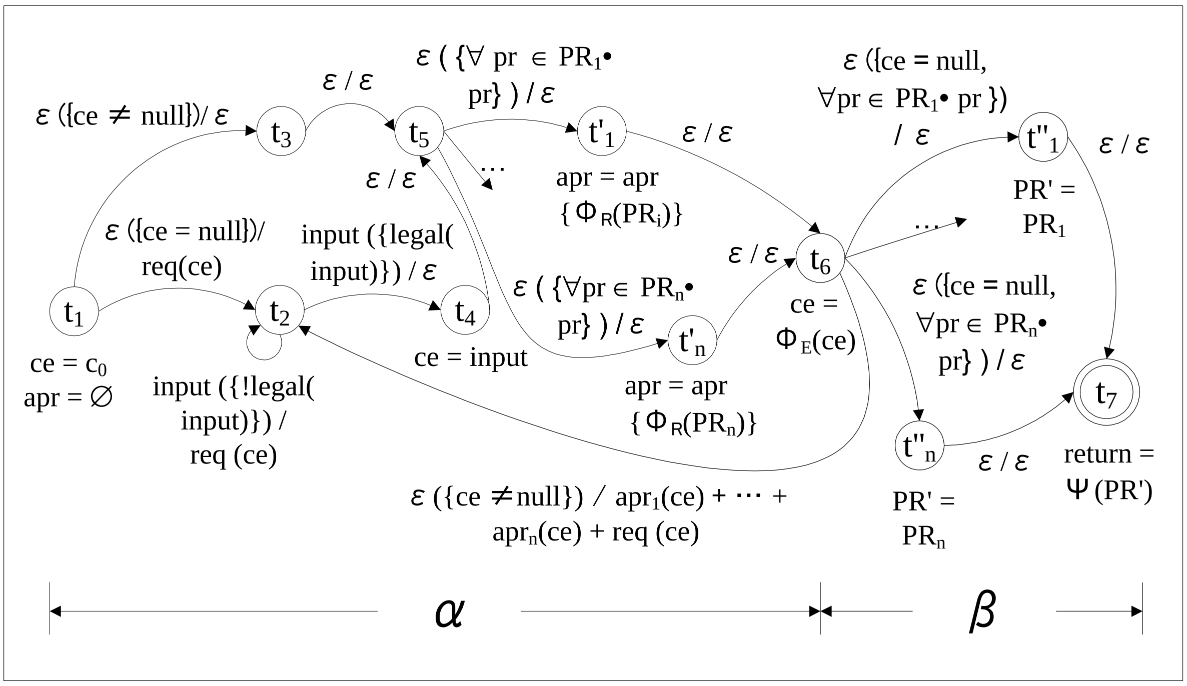

ε ({ce ≠ null})/ ε
t3
ε / ε
t5
ε ({∀ pr ∈ PR1• pr}) / ε
t'1
ε / ε
ε ({ce = null, ∀pr ∈ PR1• pr }) / ε
t''1
ε / ε
ε / ε
apr = apr {ΦR(PRi)}
PR' = PR1
ε ({ce = null})/ req(ce)
input ({legal(input)}) / ε
ε / ε
t6
ε ({∀pr ∈ PRn• pr}) / ε
ε ({ce = null, ∀pr ∈ PRn• pr}) / ε
t1
t2
t4
t'n
ce = ΦE(ce)
t7
ce = c0
apr = ∅
input ({!legal(input)}) / req (ce)
ce = input
apr = apr {ΦR(PRn)}
t''n
ε / ε
return = Ψ(PR')
ε ({ce ≠null}) / apr1(ce) + ⋯ + aprn(ce) + req (ce)
PR' = PRn
α
β

Figure 6.33. The FSM for describing the application of individual patterns

In the set AS, each FSM A_{id_i} is composed of three path sets: α, β and γ where each $\alpha_i \in \alpha$ describes step 2a of the pattern system application process, each $\beta_i \in \beta$ describes step 2b and each $\gamma_i \in \gamma$ describes the detection and correction of one kind of illegal input. Figure6.33 shows A_{id_i} where $apr_i(ce) \ \in apr$ indicates one of the constraints on element ce in apr, \$a + b$ means displaying both a and b (such as $apr_1(ce) + ... + apr_n(ce) + req(ce)$ meaning the request for specifying ce under the requirement of satisfying the listed constraints on ce), $legal(exp)$ denotes that expression exp does not violate the grammar of the used formal notation.

Paths between states t_1 and t_6 form set α where transition $t_1 \rightarrow t_3$ reflects the situation where ce has already been specified before the application of the pattern and transition $t_2 \rightarrow t_4$ denotes receiving the value to be assigned to ce from the developer. Transitions $\{t_5 \rightarrow t'_1, ..., t_5 \rightarrow t'_n\}$ determine the rule to be applied to infer satisfied constraints based on the specified ce. Paths between states t_6 and t_7 form set β where transitions $\{t_6 \rightarrow t''_1, ..., t_6 \rightarrow t''_n\}$ determine the target formal expression based on the values assigned to the elements. Paths in γ are created to handle exceptions when traversing the paths in both α and β. Loop $t_2 \rightarrow t_2$ is an example path of γ, which means that if the input on state t_2 violates the grammar of the formal notation, the developer will be asked to modify the input until it is recognized as a legal one.

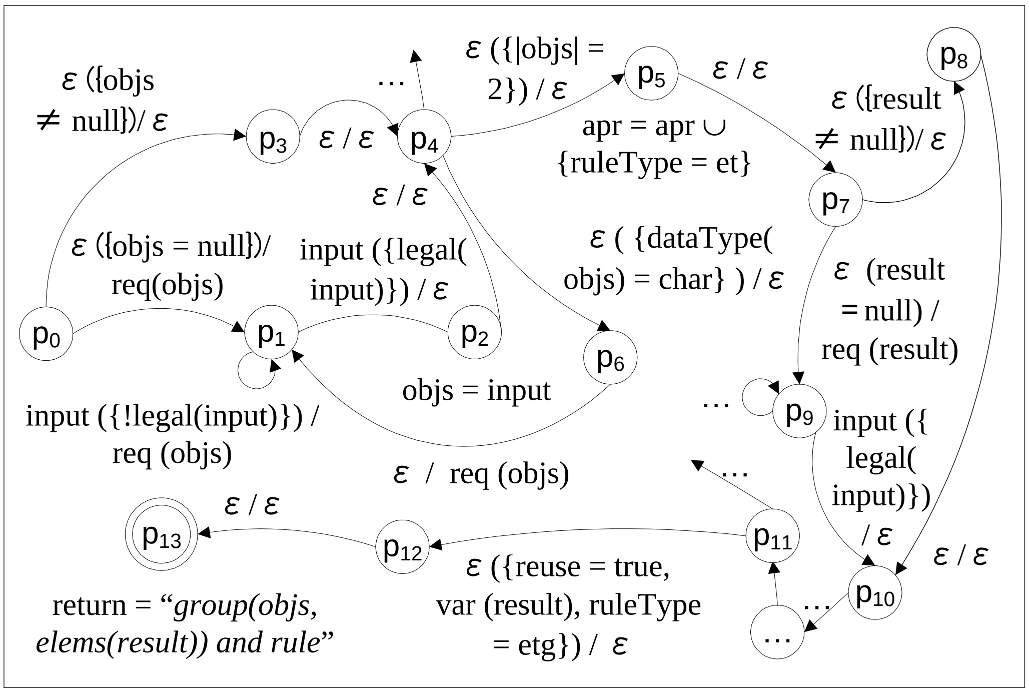

ε ({objs ≠ null})/ ε
p3
ε / ε
p4
…
ε ({|objs| = 2}) / ε
p5
ε / ε
p8
ε ({result ≠ null})/ ε
apr = apr ∪ {ruleType = et}
p7
ε / ε
ε ({objs = null})/ req(objs)
input ({legal(input)}) / ε
ε ({dataType(objs) = char}) / ε
ε (result = null) / req (result)
p0
p1
p2
p6
objs = input
…
p9
input ({!legal(input)}) / req (objs)
input ({ legal(input)}) / ε
ε / req (objs)
…
ε / ε
p13
p12
p11
ε / ε
ε ({reuse = true, var (result), ruleType = etg}) / ε
p10
…
return = "group(objs, elems(result)) and rule"
…

Figure 6.34. The FSM describing the application of the pattern "sort"

Figure 6.34 shows an example FSM A_{sort} in AS which describes the application process of the pattern *sort*. It satisfies: $\{p_0 \to p_3(p_1 \to p_2) \to p_4 \to p_5(p_6 \to p_1 \to ...) \to p_7 \to p_8(p_9) \to p_{10} \to ...\} \subseteq \alpha_{A_{sort}} \wedge \{p_{11} \to p_{12} \to p_{13}\} \subseteq \beta_{A_{sort}} \wedge \{p_1 \to p_1, p_9 \to p_9\} \subseteq \gamma_{A_{sort}}$.

Each path in $\alpha_{A_{sort}}$represents an interactive way to specify the four elements of the pattern *sort* based on the rules in the Φ item of the pattern. In $\beta_{A_{sort}}$, each path reflects a rule in the $\mathbf{\Psi}$ item and sets the formal result as the expression generated by applying the rule. If there exist informal parts in the formal result, each informal part is interpreted as the value generated by lower-level FSMs. For example, the formal expression generated on p_{13} includes "group", which indicates the informal part "$group(objs, elems(result))$" will be replaced by the value generated by traversing the FSM A_{group} with element information $(objs, elems(result))$.

6.3 Summary

In this chapter, we described a method for representing the knowledge in the *specification pattern system.* The goal of the method is to make the knowledge exposed to the developer easier to understand and the knowledge used by machines easier to be accessed and manipulated automatically. Therefore, two representations are proposed to describe the two kinds of knowledge. We have presented each of them in detail and shown the merits of adopting them.

In the next chapter, we will present how to formalize requirements into formal specifications based on the pattern knowledge represented in attribute tree and HFSM. Several example functions are adopted to illustrate the formalization process.

Chapter 7

Prototype tool for supporting the pattern-based approach

The main goal of our pattern-based approach is to support computer-aided formalization of software requirements. To validate the approach and demonstrate its efficiency, we implement it into a prototype tool that implements the approach. It interacts with the developers to derive necessary function details of the intended requirements and transformed the derived requirement into formal specifications.

We will first describe the design of the tool, as well as the involved components, and present some implementation details. Then the interface of the tool is shown, through which the main functions of the tool are illustrated. By presenting a case study on formalizing an example function, we show how the tool works for requirements formalization.

7.1 Tool design

Figure 7.35 shows the outline of the tool that is composed of four components:

- *specification pattern knowledge* stored in a XML file
- *knowledge extractor* for retrieving appropriate knowledge from the XML file
- *guidance generator* for transforming the retrieved knowledge into explicit guidance that asks for the response from the developer
- *preprocessor* for collecting input from the developer and processing it for *knowledge extractor*

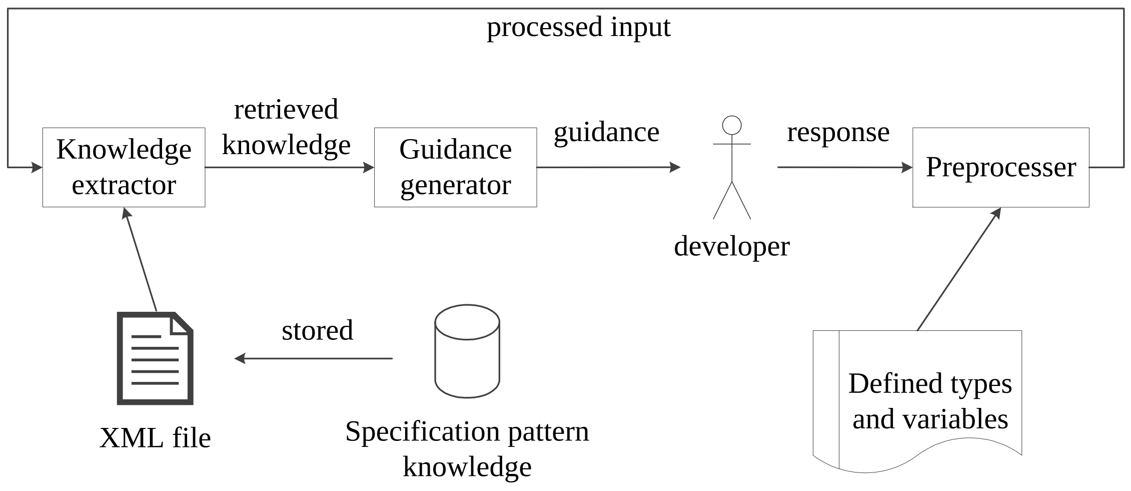


Figure 7.35. The design of the tool for supporting requirements formalization

When supporting the formalization of an intended requirement, *knowledge extractor* retrieves appropriate knowledge from the XML file that stores the *specification pattern knowledge*. The retrieved knowledge is then used by *guidance generator* to produce comprehensible guidance. By following the produced guidance, the developer is expected to respond to the tool. After receiving the input response, *preprocessor* analyzes and processes it within the context of the defined types and variables (The tool is executed on the assumption that all the necessary types and variables are already defined since the data type declaration method has not been implemented). The processed input information is used by the *knowledge extractor* to retrieve new knowledge from the XML file for producing new guidance. Such interactions continue until the target formal expression is generated.

Specification pattern knowledge is described in the previous section and we will explain the mechanisms of the other three components respectively.

7.1.1 Knowledge extractor

Knowledge extractor determines the specific knowledge to be applied according to the values of a set of state variables that reflect the attributes of the requirement under formalization. For each selected specification pattern, it retrieves knowledge from *derivation knowledge* for requirements derivation and extracts knowledge from *transformation knowledge* for requirements translation. Specifically, when supporting requirements formalization with a pattern p, *knowledge extractor* works in the following five modes for retrieving knowledge in different formalization stages.

- At the beginning of the requirements derivation stage, *knowledge extractor* first accesses to the attribute tree TR of p and obtains the root node rn of TR, the left-most child node e_1 of rn and the subtree of the node e_1. It then accesses to the clarification rule repository of e_1 and obtains the top-level rule set TRS in the repository. By evaluating the premise constraint of each clarification rule in TRS, the activated rule is found. A new constraint pr is derived by applying the activated rule. Finally, *knowledge extractor* outputs the root node rn, the subtree of the node e_1 and the newly derived constraint pr.

- During the clarification process for an attribute e_i at any level, *knowledge extractor* accesses to the rule repository of e_i and obtains one of the low-level rule sets RS according to the previously activated rule. After evaluating the premise constraint of each rule in RS, the activated rule is found and a new constraint pr is obtained and generated by applying the activated rule. If pr sets a constraint on an attribute of *Req* type where specification pattern p' is involved to specify the attribute, *knowledge extractor* retrieves knowledge in the first mode from the *derivation knowledge* of p'.

- After the clarification of a low-level attribute of *Req* type, *knowledge extractor* goes back to the clarification rule repository of its parent attribute and generates the activated rule.

- When the clarification of a node e_i in TR is finished, *knowledge extractor* accesses to TR and returns the subtree of the node e_{i+1} and the activated rule in the top-level rule set in the clarification rule repository of e_{i+1}.

- After all the attributes represented in TR is clarified, *knowledge extractor* accesses to the *transformation knowledge* of p and implements the algorithm represented as the previously introduced function $reqTransform$. The returned formal expression is generated as the retrieved knowledge.

  The algorithm for utilizing the pattern knowledge represented in HFSM is shown in Figure 7.36. It starts from the initial state of A_{root} and displays guidance by traversing the states of the HFSM according to actual inputs from the developer.

```
Algorithm        Utilization of the HFSM

cf = $A_{root}$; cs = $q0_{cf}$; input = return = null;
while(cs $\notin F_{cf}$ || states is not empty){set variable input as the input from the user;
   if(cs $\notin F_{cf}$){ns = o = null;
      foreach (i, G) in $Acc_{cf}$(cs){
         if(i == input && $\forall g \in G \cdot g$){ns = $\delta_{cf}$(cs, (i, G)); o = $\lambda_{cf}$(cs,(i, G));}
      if($\sigma_{HF}$(cs) $\neq \emptyset$ || $\exists v \in V\varphi_{cf}(cs) \cdot \sigma_{HF}(v) \neq \emptyset$ || $s_{HF}(o) \neq \emptyset$){states.push(ns);
         if($\sigma_{HF}$(cs) $\neq \emptyset$)  set ns as the initial state of one of the FSMs in $s_{HF}$(cs);
         if($\exists v \in V\varphi_{cf}(cs) \cdot \sigma_{HF}(v) \neq \emptyset$){
            for each $v \in V\varphi_{cf(cs)}$ that satisfies $\sigma(v) \neq \emptyset${queue.push(v);}
           if($\sigma_{HF}$(cs) = $\emptyset$){
               set ns as the initial state of one of the FSMs in $s_{HF}(v)$ where $v \in V\varphi_{cf(cs)}$;}
         else{specify variables according to $\varphi_{cf}$(cs)}
         if($s_{HF}(o) \neq \emptyset$){queue.push(o);
            if($s_{HF}$(cs) = $\emptyset$ && $s_{HF}$(o)= $\emptyset$){set ns as the initial state of a FSM in $s_{HF}$(o);}}
         else{display o to the user;}
   else{if(return $\neq$ null ){replace the corresponding part in queue[top] with return;
         if($s_{HF}$(queue[top]) = $\emptyset$){temp = queue.pop();
            if(temp is output){display temp;}else{specify variables based on temp;}}}
   ns = states.pop();}
cs = ns; cf = A where cs $\in Q_A$;}
```

Figure 7.36. The algorithm for utilizing the HFSM

When traversing each FSM A in the HFSM, A is transferred from the current state to the next state according to input. On each non-accept state cs in A with input symbol $input$, the next state is determined by analyzing each transition in $Acc_A(cs)$. If there exists one transition labeled $i(G)/o$ where $i = input$ and all the guard conditions in G are satisfied, A will be transferred to the destination state of the transition and o will be displayed. The above process repeats until reaching the accept state.

To deal with hierarchical relations in the HFSM, the following method is adopted: given a upper-level FSM A, when encountering a component c that is interpreted by a set of lower-level FSMs LF, the current state s will be stored and FSMs in LF will be traversed. If a value is returned after all the FSMs in LF are terminated, it will be used to specify c. Then, the algorithm sets the current state back to state s and continues the traverse of FSM A.

7.1.2 Guidance generator

Guidance generator produces and displays guidance according to the knowledge returned from the *knowledge extractor*. As shown in Table 7.8, there are four major kinds of guidances corresponding to the four kinds of knowledge retrieved from the *specification pattern knowledge*.

7.1.3 Preprocessor

Preprocessor includes a set of rules for determining the semantics of the user input. Each rule transforms user input into the value of one kind of state variable that represents one kind of attribute. The obtained state variable's value will be used to retrieve knowledge for guiding the user in the next step of the formalization process.

Assume guidance g is displayed for guiding the assignment of an attribute e and the developer inputs a string to the tool as the response to g based on his understanding on the intended requirement. *Preprocessor* "understands" the meaning of the input string based on the type of e. For example, if e is of $expValue$ type, *preprocessor* will treat the input string as a formal expression representing certain system variable and transform it into the value of a state variable that represents the attribute e.

7.2 Tool implementation

The major concern when implementing the above design is the format for storing the HFSM model in the tool. Considering that XML is becoming widely used in industry for its simplicity, it is used to carry the information in the knowledge base so that the knowledge can be easier shared by other communities.

As a markup language, XML requires a set of tags to identify data with different meanings. Table 7.9 shows the XML tags for the HFSM model.

These tags solve the problem of storing all kinds of components in FSM models. To illustrate their use in the tool, the XML representation of an example FSM is given in Figure 7.37 where the left part shows the example FSM and the right part shows its XML representation.

To store the hierarchical relations among FSMs, two mechanisms are given for the two kinds of hierarchies. The first mechanism is the use of FSM names as destination states to represent the interpretation of high-level states by low-level FSMs. For example, "<dest>F1<dest>" involved in a <state> fragment of the state s means that

Table 7.8. Guidance produced from the knowledge retrieved by the *knowledge extractor*

Retrieved knowledge		Produced guidance
the root node of an attribute tree		Display a requirement tree RT initialized as a root node denoting the requirement to be formalized
the subtree of a node e_i in an attribute tree		Create a same node e_i as a child node of the root node in RT
a constraint derived by applying certain transformation rule	attribute e_i is of atomic type	If e_i is of *choice* type, ask for choosing from the displayed candidate items. Otherwise, ask for the input the intended value in a text box.
	attribute e_i is of *composite* type	Create a child node of the node e_i in RT for each low-level attributes of e_i. For each low-level attribute e_{ij} of e_i, if the constraint contains sub-constraint on the definition of e_{ij}, produce guidance according to the sub-constraint.
	attribute e_i is of *set* type	Ask the developer to add member attributes to e_i. Each time when a new member attribute is added, create a child node of e_i for denoting the member attribute and produce guidance according to the constraint on the definition of the member attributes of e_i.
	attribute e_i is of *Req* type and described by applying the pattern p	Create a requirement tree RT' with node e_i as its root node. Then produce guidance according to the knowledge retrieved from the attribute tree of p and the clarification rule repository of the first attribute node in the attribute tree.
	attribute e_i is of *union* type	Display guidance that asks for the developer to clarify the definition of e_i by choosing one of the constituent types of the *union* type
	$e_i = v_i$	Assign v_i to attribute e_i
a formal expression		Display the formal expression

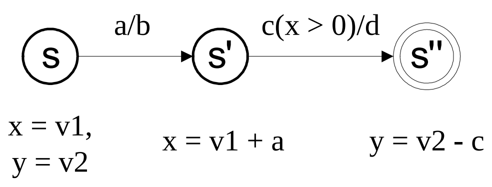
Table 7.9. Tags for identifying the elements in the FSM model

tag	**the corresponding elements in the HFSM model**
<state>	The states of each included FSM model
<transition>	transitions originating from certain state
<input>	input of a transition label
<guard>	guard condition of a transition label
<output>	output of a transition label
<dest>	destination states of transitions
<inf>	value information of state variables
<para>	variable names
<value>	the value of variables
...	...

the inner structure of s is interpreted by the FSM $F1$. The other mechanism is the use of FSM names as the value of state variables to represent the hierarchical relations between the state variables in high-level FSMs and low-level FSMs for interpreting these variables. For example, "<value>F2<value>" involved in a <inf> fragment of the variable v means that $F2$ is a *value FSM* and v is assigned with the value returned after traversing $F2$.

Moreover, in order to enable facilitate the description of the informal guidance indicated in the *expl* item of the specification patterns, several symbols are introduced as shown in Table 7.10.

When implementing the tool, the FSM model of the pattern system is stored as knowledge in a XML file based on the pre-defined XML tags. Other components extracts state and transition information from the file for implementing the knowledge retrieval algorithm and produces comprehensible guidance with informal explanations attached to symbols in output.



```
<state>
  <name>s</name>
   <inf>
    <para>x</para>
    <value>v1</value>
  </inf>
  <inf>
    <para>y</para>
    <value>v2</value>
  </inf>
  <transition>
    <input>a</input>
    <guard></guard>
    <output>b</output>
    <dest>s'</dest>
  </transition>
</state>
```

```
<state>
  <name>s'</name>
  <inf>
    <para>x</para>
    <value>v1+a</value>
  </inf>
  <transition>
    <input>c</input>
    <guard>x>0</guard>
    <output>d</output>
    <dest>s"</dest>
  </transition>
</state>
```

```
<state type = "accept">
  <name>s"</name>
  <inf>
    <para>y</para>
    <value>v2-c</value>
  </inf>
</state>
```

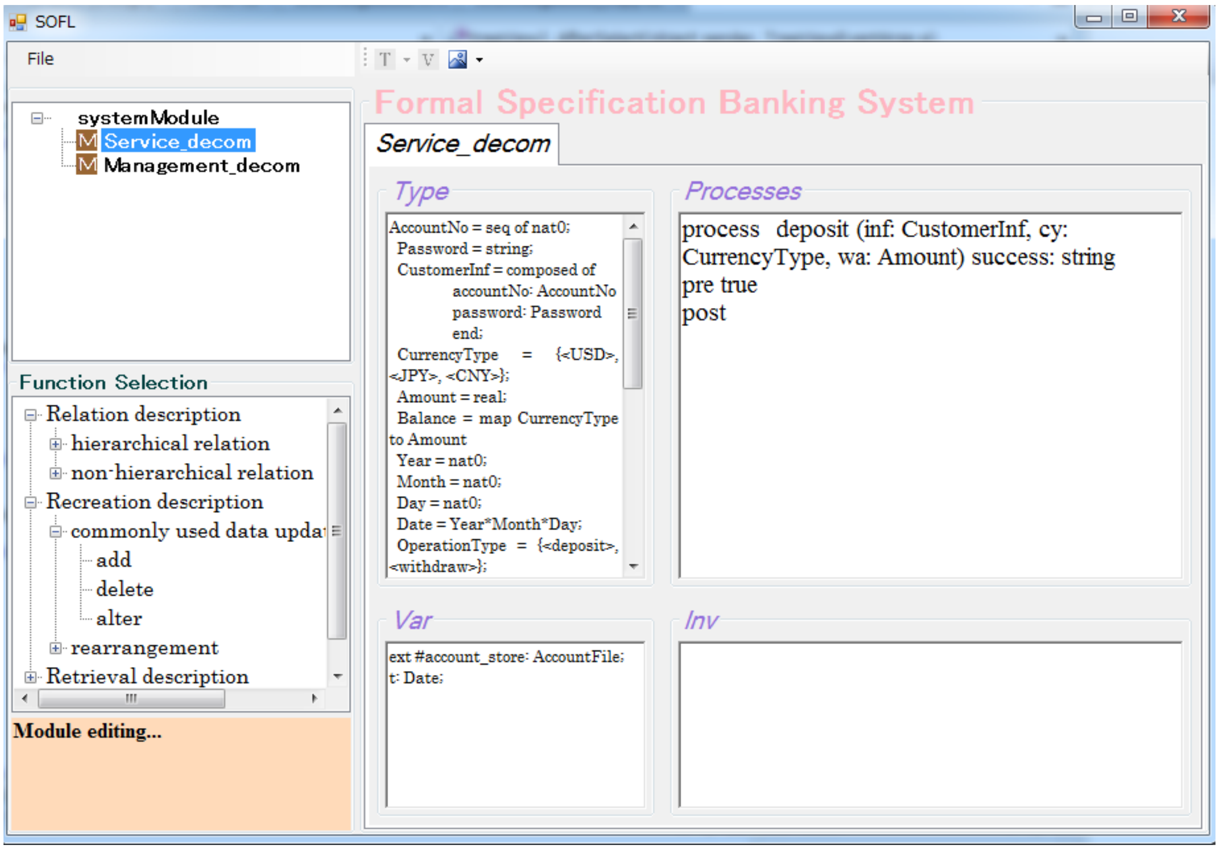
Figure 7.37. An example FSM and its XML representation

Table 7.10. Symbols involved in FSM models

symbol	definition
$\sum S$	Providing items in set S for the designer to choose from
$\&item_i$	Selection of item $item_i$
$\#k$	Asking for pressing key k
$req(x)$	Element or variable x is required to be designated with a value
$legal(i)$	Input symbol i is written in defined variables and formal notations
$patterns$	The variable indicating all the patterns in use
$pattern$	The variable indicating the pattern currently being applied
$elems$	A variable of sequence type that holds the values of the elements of $pattern$
$\#mM.v$	The value of variable v in module M
$formalExp$	The variable that holds the generated formal result
...	...

7.3 Tool interface

Although the underlying theory of the tool is language-independent, a specific formal notation needs to be chosen when implementing the tool. Due to our expertise, we choose SOFL as an example formal notation and implement the tool to support requirements formalization during the construction of SOFL formal specifications. Figure 7.38 shows a snapshot of the main frame of the tool being executed for supporting the writing of the formal specification of a banking system.



```
AccountNo = seq of nat0;
  Password = string;
  CustomerInf = composed of
            accountNo: AccountNo
            password: Password
            end;
  CurrencyType    =    {<USD>,
<JPY>, <CNY>};
  Amount = real;
  Balance = map CurrencyType
to Amount
  Year = nat0;
  Month = nat0;
  Day = nat0;
  Date = Year*Month*Day;
  OperationType  =  {<deposit>,
<withdraw>};
```

```
process  deposit (inf: CustomerInf, cy:
CurrencyType, wa: Amount) success: string
pre true
post
```

```
ext #account_store: AccountFile;
t: Date;
```

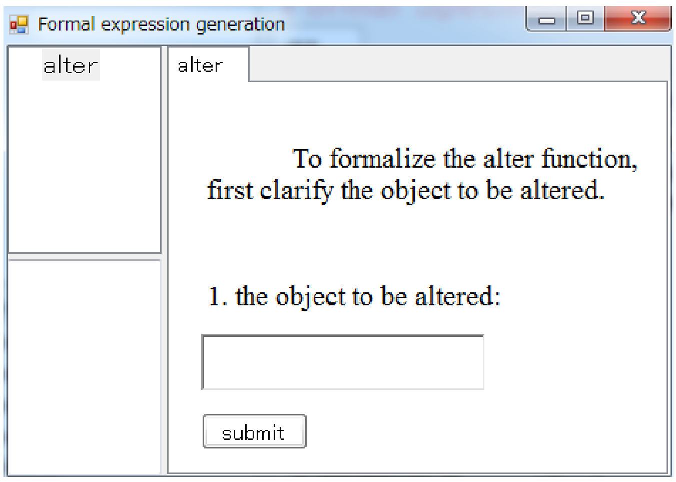
Figure 7.38. The main frame of the tool

The tree structure on the top left reflects the architecture of the specification where each node indicates a module (In SOFL, a formal specification consists of a set of modules. Each module describes an relatively independent function by a set of inter-related processes. Each process describes an operation producing outputs from inputs in terms of pre- and post-conditions). High-level modules are decomposed by attaching child nodes representing low-level modules. For the banking system, the top level module *systemModule* is decomposed into two low-level modules *Service_decom* and *Management_decom*. The module *Service_decom* describes the banking services provided by the system for the customers owning authorized accounts and the module *Management_decom* describes the operations for analyzing and maintaining the system information. The right part of the interface is used to edit the content of the selected module where *Type* denotes the declaration of custom data types, *Var* denotes the declaration of specification variables, *Processes* denotes the collection of processes describing various operations in the module and *Inv* denotes the collection of invariants each expressing a property that must be sustained throughout the entire specification. When editing a module, its types and variables need to be first declared and the tool will use these pieces of information to guide the formalization of pre- and post-condition of each process, as well as invariants.

Formalizing a pre/post-condition or an invariant starts from manually analyzing and decomposing the pre/post-condition into a set of basic functions. For each basic function, a specification pattern is chosen by selecting a function from the tree structure on the bottom left of the main frame. This tree structure categorizes all the specification patterns according to the functions they can be used to formalize to facilitate correct selection. One can also retrieve the explanation of each node to confirm whether it matches the intended requirement. Starting from the top level of the hierarchy, the developer is required to select a sub-category on each level until reaching a pattern. It is not difficult to find the right pattern because of three reasons. First, pattern names are written in natural language and designed to be distinguishable from each other on the semantic level. Second, the patterns are organized by categories at different levels and the developer only needs to deal with one category or sub-category at a time. Third, the explanation on the patterns describe their usage in more details and can help confirm the selection decision.

For each selection decision, a new frame will be popped up as the medium to derive necessary function details of the intended basic function. Assume that the pattern *alter* is selected which is used to formalize the functions of altering existing data items of system variables, a frame as shown in Figure 7.39 will be displayed. Its left part shows the requirement tree which is initialized as a root node denoting the selected pattern. It will be gradually constructed automatically during the requirements derivation process. The right part shows the guidance for clarifying the selected node in the requirement tree and receives the response from the developer.

All the guidances are numbered to illustrate their display order. According to the attribute tree of the pattern



To formalize the alter function, first clarify the object to be altered.

1. the object to be altered:

Figure 7.39. The snapshot of the supporting tool

alter shown in Figure 6.20, the constraint *obj*, i.e., the object to be altered, needs to be first clarified. Therefore, the top-level rule sets in the clarification rule repository of *obj* is applied to produce the first guidance that asks for the assignment of *obj*. After the developer fills in the displayed text box using declared variables and presses the submit button, the tool will send the provided information to the *preprocessor* and activate the *knowledge extractor* to retrieve knowledge from the XML file that stores the *specification pattern knowledge* according to the processed input. The *guidance generator* will then produce new guidance based on the retrieved knowledge for further interactions. When adequate function details are derived, a formal expression will be generated and displayed on the frame. It is allowed to be modified and copied to the main frame as the formalization result of the corresponding pre/post-condition or invariant.

7.4 Case study

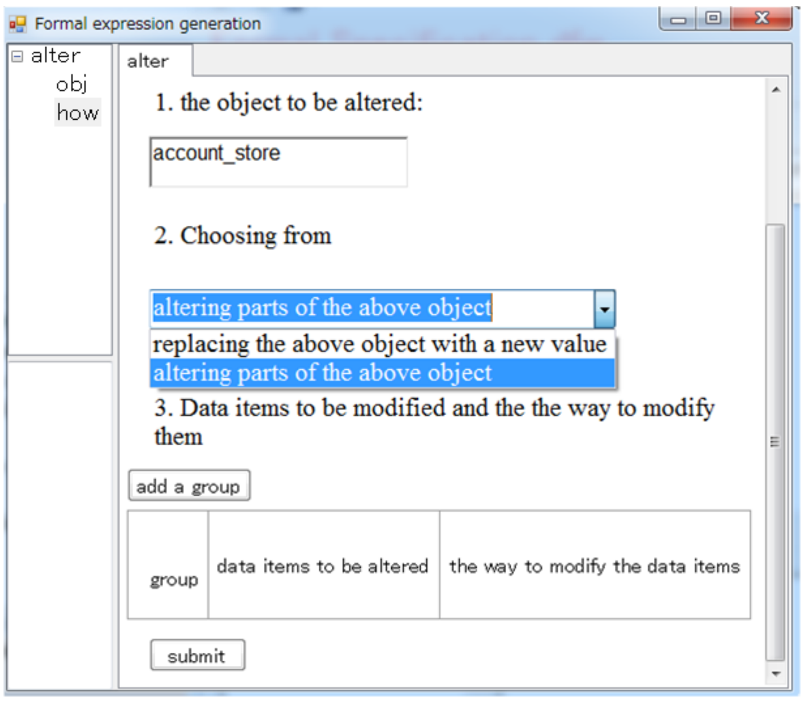
To demonstrate how the tool works, this section presents the formalization process of the *deposit* function of the banking system. The process *deposit* is created in the previously introduced module *systemModule* to describe this function (Figure 7.40 shows the types and variables defined in the module). It contains three inputs and one output where input *inf* denotes the identity of the customer who deposits money to the system, *cy* denotes the type of the currency to be deposited, *wa* denotes the amount of the currency to be deposited and *success* denotes the message declaring the success of the deposit operation. The pre-condition is simply set as *true* and the writing of the post-condition needs guidance. According to the semantics of the *deposit* operation, the post-condition should describe the way to update the account information after *deposit* operation. Since it is an alter function, pattern *alter* is selected to assist the formalization of the post-condition and a frame in Figure 7.39 will be displayed.

```
type
 AccountNo = seq of nat0;
 Password = string;
 CustomerInf = composed of
                  accountNo: AccountNo
                  password: Password
                 end;
 CurrencyType = {<USD>, <JPY>, <CNY>};
 Amount = real;
 Balance = map CurrencyType to Amount
 Year = nat0;
 Month = nat0;
 Day = nat0;
 Date = Year*Month*Day;
 OperationType = {<deposit>, <withdraw>};

 Transaction = composed of
                  date: Date
                  operationType: OperationType
                  currencyType: CurrencyType
                  amount: Amount
                 end;
 AccountInf = composed of
                  balance: Balance
                  transactions: set of Transaction
                 end;
 AccountFile = map CustomerInf to AccountInf;

var
 ext #account_store: AccountFile;
 ext #today: Date;
```

Figure 7.40. The defined types and variables in the module $systemModule$

Figure 7.41 demonstrates the interaction process led by the first three pieces of guidance. The first guidance requires the clarification of the attribute *obj* in the requirement tree. According to the definition of *obj*, it should be specified with a system variable represented in SOFL. Denoting the data store for carrying the account information, variable *account_store* is input as the response to this guidance. With the analysis result on the given variable, the tool starts to retrieve knowledge for assigning attribute *how* and the second guidance is displayed that provides two items to choose from. Considering that the *deposit* operation only updates the account information of the customer identified as inf, rather than replacing the whole *account_store* with a new value, the second item is chosen. After receiving the selection decision, the tool "understands" that the user intends to assigned how a value of type d_1 and produces the third guidance in terms of a table. Each row of the table *groupi* indicates one kind of data items needed to be modified in *account_store* (denoted as attribute *data*) and the way to modify them (denoted as attribute *oper*). This guidance should be responded by adding new rows to the table according to the intended requirement.

The button "add a group" at the top of the table is provided for adding new rows. When it is clicked to add a row, guidance for clarifying the new row will be displayed. Figure 7.42 shows the produced guidance when the "add a group" button is pressed (The response to the guidance is also shown which will be explained later). Item



1. the object to be altered:

account_store

2. Choosing from

altering parts of the above object

replacing the above object with a new value
altering parts of the above object

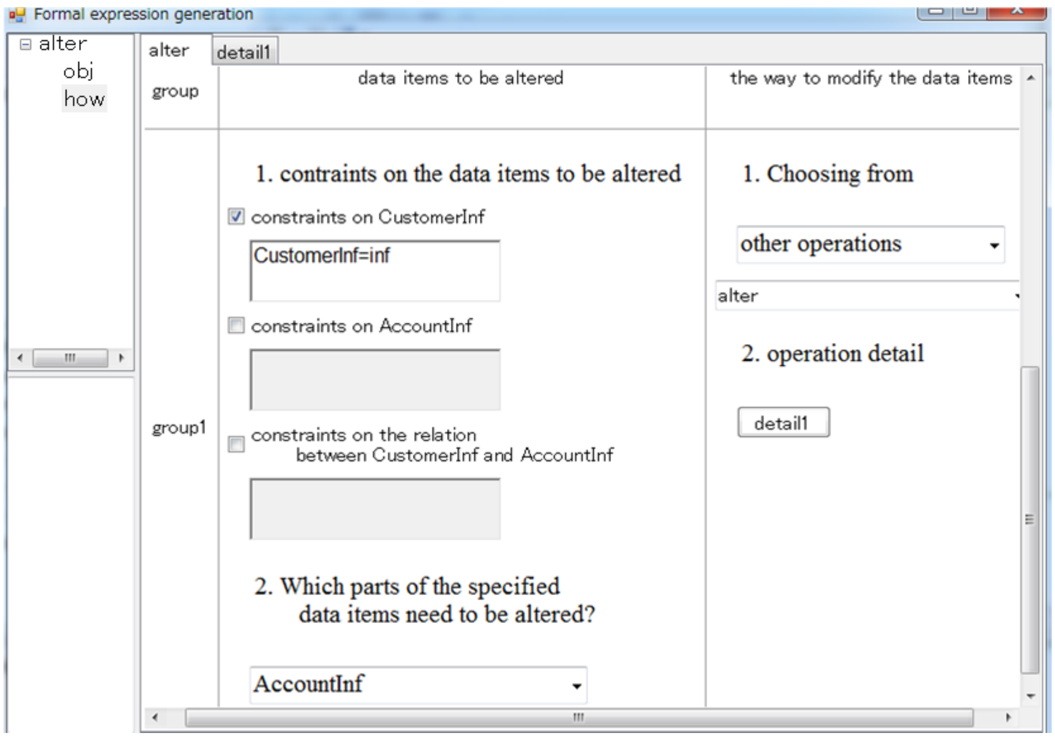
3. Data items to be modified and the the way to modify them

group	data items to be altered	the way to modify the data items

Figure 7.41. The snapshot of the supporting tool

data of the new row is guided to be clarified from two aspects. The first aspect is the constraints on the data items to be altered, i.e., the description on what kind of data items are required to modify. It includes three optional items and should be clarified by specifying at least one of them. In the *deposit* function, the data item to be altered in $account\_store$ is the information of the customer identified as inf. Therefore, only the first item should be selected and clarified as one constraint on the corresponding $CustomerInf$: $ConstraintInf = inf$ ($CustomerInf$ is the domain of the variable $account\_store$ defined as a mapping). The second aspect is the parts of the data item specified in the first aspect are needed to be modified. Three candidate items are provided: $CustomerInf$, $AccountInf$ and *both*. Considering that the *deposit* function does not rewrite the customer information and only alters the account information of the customer identified as inf, item $AccountInf$ should be selected.

Item *oper* of the new row is first specified by choosing from items "replace $AccountInf$ with a new value" and "other operations". According to the semantics of the function *deposit*, the information of the target account is updated by modifying its balance and transaction parts, rather replacing itself with a new account information. Therefore, the second item should be selected and a new guidance is consequently produced which asks the selection of the specific operation for altering the account information. The candidate operations are provided according to


Formal expression generation

alter
obj
how

alter | detail1

group | data items to be altered | the way to modify the data items

group1

1. contraints on the data items to be altered

☑ constraints on CustomerInf

CustomerInf=inf

☐ constraints on AccountInf

☐ constraints on the relation between CustomerInf and AccountInf

2. Which parts of the specified data items need to be altered?

AccountInf

1. Choosing from

other operations

alter

2. operation detail

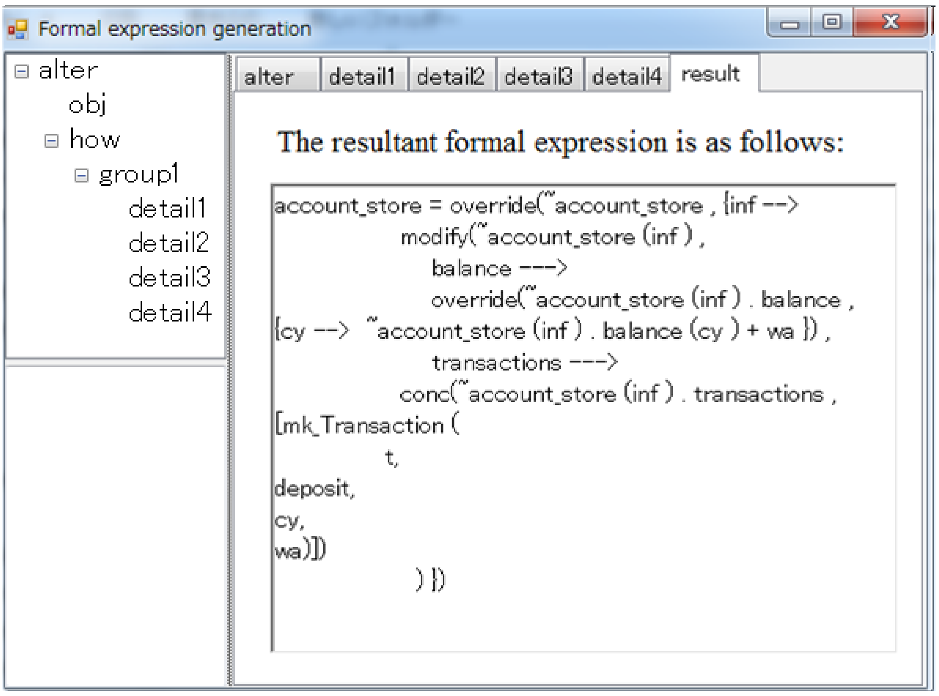
detail1

Figure 7.42. The snapshot of the supporting tool

the patterns included in the *recreation* category. By analyzing the intended requirement, the selection decision on alter operation can be made. To clarify the details of the alter operation, the guidance numbered 2 is produced with a button "detail1", and a new page for guiding such a clarification process is created and linked to the button.

After responding to the produced guidance by filling out each graphical components according to the semantics of the *deposit* operation, the developer will be guided to further clarify the attributes for composing the *deposit* operation. Such kind of interactions repeat until all the attributes in the attribute tree of the pattern *alter* are assigned. With a complete requirement tree that represents all the necessary function details of the *deposit* operation, a formal expression is generated as shown in Figure 7.43. It can be manually revised and copied to the main frame as the post-condition of the process *deposit*.

This case study shows that the pattern-based approach can effectively support requirements formalization and HFSM representation successfully supports the automatic use of the pattern knowledge. The tool separates the tasks of clarifying requirements and formally representing the clarified requirements by allowing human decisions on function details and automating the translation of the function details. Developers without formal notation



The resultant formal expression is as follows:

```
account_store = override(~account_store , {inf -->
                modify(~account_store (inf ) ,
                    balance --->
                    override(~account_store (inf ) . balance ,
{cy -->  ~account_store (inf ) . balance (cy ) + wa }) ,
                    transactions --->
                conc(~account_store (inf ) . transactions ,
[mk_Transaction (
              t,
deposit,
cy,
wa)])
                  ) })
```

Figure 7.43. The snapshot of the supporting tool

expertise are also able to write formal expressions using the tool, since the guidance is given in natural language and can be easily followed.

Notice that the above interaction process involves the use of defined variables since they are the only representation of the system objects that can be understood by both the tool and the user. Thus, developers should be clear about the relations between the defined variables and the real system before using the tool to formalize system requirements. One way to avoid the use of defined variables is to assign them with informal semantics. But whether this is necessary needs further investigation.

7.5 Summary

In this chapter, we have described the prototype tool that implements the pattern-based approach. The design of the tool is first given which specifies its architecture and demonstrates how the involved components cooperate to perform the required functionality. It also includes the explicit description on each component. Then some implementation details are presented where the most critical one is the format for storing the HFSM representation of the pattern knowledge in the tool. Finally, the major functionality of the tool are introduced through its interface and a case study on the tool is presented to illustrate these functionality in real practice.

In the next chapter, we will present the experiments held on the prototype tool. The goal of the experiments is to check whether the tool, as well as the pattern-based approach, can effectively support the requirements formalization process and help us explore the improvements that need to be done in our future work.

Chapter 8

Experiment

To evaluate the effectiveness of the pattern-based approach, two controlled experiments on the supporting tool have been conducted.

These two experiments involve subjective evaluation and the result is affected by the ability and bias of the participants. This stems from the fact that software engineering includes many subjective factors such as usability. When experiments on software engineering techniques are designed, subjective evaluation is inevitable if the subjective factors of these techniques need to be estimated [99] [100] [101]. In our case, for example, we can only depend on the feedback of the participants to evaluate whether the interface and the provided guidance are easy to use.

Because of this reason, it is hard to formally prove that our experiment result exactly reflect the performance of our tool in real practice. In order to create an experimental environment that is similar to the real settings, we choose students as the participants of our experiment since their experience in requirements formalization is similar to practitioners in software industry. Therefore, the experiment result can demonstrate the fundamental properties of the proposed approach and the prototype tool.

8.1 Experiment for investigating the ability and usability of the tool

In the first experiment, we invited our master students to use the supporting tool to formalize the functions of several typical software systems. These students have studied the SOFL formal language for one or two years and have written two or three SOFL formal specifications. They are able to read and even analyze a SOFL formal specification but still not experienced enough to express their envisioned functions in SOFL formal notations efficiently.

There are six software systems to be formally specified: Hotel reservation system, Banking system, E-ticket system, Suica card system, Library information system and Online shopping system (For concise illustration, we will use H, B, E, S, L, O as the abbreviation of these six systems respectively in the following presentation). Each student is asked to write the formal specification of one of these systems using the tool. For the purpose of assessing the application domain of the patterns, manual formalization is not allowed, i.e., all the pre- and post-conditions are required to be written under the guidance of the tool.

Table 8.11 shows the result of the experiment. It summaries the collected data for each formal specification and its construction process. The second column indicates the number of the included processes and the third column records the number of the patterns applied for writing these processes. The fourth column denotes the percentage of the guidance that is easy to follow where ss denotes the number of pattern selection decisions easy to be made, s denotes the total number of the pattern selection activities, sg denotes the number of guidelines easy to understand and g denotes the total number of the displayed guidelines. It represents the simplicity of using the tool in requirements formalization, including the simplicity of selecting appropriate patterns and the simplicity of interacting with the given guidance.

After interviewing the participants, we found that the pattern system can cover all the functions in these six systems. The provided categorization tree facilitates pattern selection and the distinct pattern and category names give little chance to wrong selections. The major difficulty is the decomposition of the intended functions into basic functions that can be formalized by patterns. They suggest the design of more abstract patterns for specific systems to further facilitate pattern selection. Designing such kind of patterns needs technical support from domain experts and we will extend our framework along this line based on the foundation proposed in this thesis. We also found that most of the participants cannot fully understand the representation of the provided guidance when formalizing the first several functions. But once getting familiar to the guidance representation through formalizing the first several functions, they can independently interact with the prototype tool to formalize the rest of the functions.

The last column reveals the number of errors explored in the formal specification. Most of these errors are caused by the misunderstanding of the displayed guidance when formalizing the first several functions. As more functions are formalized, fewer errors are made.

Table 8.11. The result of the first experiment

Software system	**Number of processes**	**Number of applied patterns**	$((ss/s) + (sg/g))/2$ **(%)**	**Number of errors**
H	53	10	89%	13
B	49	9	85%	11
E	50	9	89%	10
S	53	11	87%	13
L	55	12	90%	10
O	60	12	85%	11

Although this result cannot lead to the conclusion that any requirement can be formalized by a set of our patterns, it does demonstrate that the proposed approach is able to support computer-aided formalization of commonly used functions.

8.2 Experiment for evaluating the effectiveness of the tool

In the second experiment, a class of undergraduate students who have received training on SOFL for only one semester are invited to formalize the requirements of a banking system. This system consists of two parts: a sub-system for providing banking services to customers and the previously introduced bank data analysis system for managers. We selected 11 processes from the formal specification of the banking system and replaced their post-conditions with informal explanations on the behaviors of the processes. The participated 76 students are divided into two groups. Each student in group 1 is asked to manually write the post-conditions of the 11 processes and the students in group 2 formalize the 11 processes in terms of post-conditions by using our prototype tool. All the students are required to record the time they spent for formalizing each process behavior.

After collecting the materials provided by the students and reviewing the submitted formal specifications, the result of the experiment is organized in Table 8.12. As can be seen from the table, the average time for formalizing

Table 8.12. The result of the second experiment

Process behavior	**Average time for formalization**		**Average number of errors**	
	group 1	group 2	group 1	group 2
customer authorization	4.5 min	1min	2	0
deposit	16.7min	14min	6	2
withdraw	7.6min	4min	5	2
currency exchange	10.7min	5.2min	7	1
information display	24.5min	8.4min	9	1
transfer	13min	6.5min	4	0
manager authorization	0.4min	0.8min	1	0
transaction analysis	15min	4.2min	5	2
balance analysis	8.5min	3.8min	6	2
global transaction analysis	19.7min	7.9min	8	3
global balance analysis	14min	7.1min	9	1

each function in group 2 is less than that in group 1 while the average number of errors found in the formalization result of each function in group 1 is more than that in group 2. This demonstrates that the tool can help formalize requirements more efficiently and enhance the quality of the resultant formal expressions. Besides, as the complexity of the intended function increases, more time will be saved. For example, process behavior *information display* is more complex than *withdraw*. Formalizing *information display* with the tool saves more time than *withdraw*. This is because as the complexity of the function increases, students in group 1 have to spend more time on both clarifying and transforming the details of the function. By contrast, students in group 2 only need to respond to more guidances for requirements clarification and the tool will handle the rest of the work.

There are also some data demonstrating the same conclusion with the first experiment. Although the complexities of the functions *deposit* and *withdraw* are the same, there is a large difference between the average times for formalizing them in group 2. An important reason is that the students have deepened their understanding on the meaning of the guidance produced by the tool when formalizing the function *deposit*, which enables them to respond to the tool more quickly and speeds up the interaction process for formalizing the function *withdraw*. Therefore, a course needs to be designed to train the potential users to help them use the tool more efficiently.

Another interesting phenomenon is that formalizing a behavior by reusing the formal representation of a similar one can promote the efficiency. For example, the formal representations of the behaviors *transaction analysis* and *balance analysis* are similar and the formalization of *balance analysis* can be easily done by modifying certain parts of *transaction analysis*'s formal representation. As can be seen from the table, the average time for formalizing *balance analysis* is much less than *transaction analysis* in group 1 although their complexities are almost the same. This phenomenon indicates a way to enhance the efficiency of the tool where some "standard" formal fragments can be designed to facilitate the formalization of the similar functions.

Note that the students participating in both experiments are non-experts in requirements formalization. We chose them because the proposed approach mainly aims to support non-experts and the experiment result can reflect the effectiveness of the approach.

Nevertheless, it is also important to know the approach's performance on supporting experts. We have discovered some features according to our experience in using the prototype tool, although they are not proved by large-sized experiments yet. For simple functions, manual formalization is more efficient than tool-supported formalization. The reason is that experts have formed their own patterns in mind and can quickly write formal expressions based on these patterns. For sufficiently complex functions, manual formalization becomes time-consuming and error-prone even for experts, and the prototype tool can largely reduce the time cost and enhance the quality of the formalization result. Another discovery is that the amount of interactions is appropriate for non-experts but needs to be reduced for experts. One solution is to design the pattern knowledge on different levels for assisting different users, which is part of our future work.

8.3 Summary

In this chapter, we have presented two experiments on the developed prototype tool for supporting the pattern-based approach. The first experiment aims at investigating the ability of the tool in formalizing the requirement of several example software products and the usability of the tool in real practice. The experiment result shows that the tool is easy to use once the developer gets familiar to the representation of the produced guidance. The second experiment compares manual requirements formalization and tool-supported requirements formalization to evaluate the effectiveness of our prototype tool. The experiment result demonstrates that the tool is able to help requirements formalization more efficiently and reduce the errors in the resultant formal specifications.

The next chapter is the last chapter of the thesis which gives the conclusion on our research work and points out the future work directions.

Chapter 9

Conclusion and future work

9.1 Conclusion

Formalizing informal requirements into formal specifications significantly improve the accuracy of the requirements and help deepen the understanding of the envisioned system. However, this activity requires high skills for abstraction and the use of formal notations, which remains a challenge to most of the practitioners.

To assist practitioners in formalizing requirements, this thesis proposes a pattern-based approach to guide the clarification of requirements and representation of the clarified requirements in formal expressions. A specification pattern system is pre-defined in this approach to include a set of patterns categorized in a hierarchy according to the functions they are used to formalize. A method for guiding the requirements formalization by applying the *specification pattern system* is given. It only requires the developers to make decisions on function design issues and handles the rest of the formalization work.

Attribute tree and HFSM are adopted to represent the pattern knowledge. The former facilitates developers' understanding on the structure of the intended requirement while the latter facilitates the utilization and maintenance of the pattern knowledge. Algorithms are given to utilize the pattern knowledge represented in these two languages.

A prototype tool that implements the proposed pattern-based approach is developed and described. We explain the underlying theory of the tool and illustrate its major functions through a case study. It demonstrate the validity of the introduction of specification patterns for requirements formalization and the effectiveness of the proposed representation method in using, storing and managing the pattern knowledge. Through two experiments, we have shown that the proposed approach is able to support computer-aided formalization of requirements and even developers without formal notation expertise are able to achieve qualified formal specifications using the tool. When writing formal expressions, practitioners are only required to focus on the design of the relevant functions and the tool will handle the rest of the work. Besides, the tool can help formalize requirements more efficiently and enhance the quality of the resultant formal expressions.

9.2 Future work

9.2.1 Future research on the pattern-based approach

Our first experiment applies the prototype tool to several software systems and demonstrates that it is able to tackle most of the commonly used functions. But the application to large-scale systems is still needed to improve the pattern knowledge. We plan to carry out more large-scale experiments involving both non-experts and experts in the future to observe the effectiveness of our approach in supporting engineers with different levels of formal specification writing skills. For non-experts, we plan to invite our partners from industry to use the tool in their real projects, and collect the result and feedback to improve the approach. For experts, we plan to invite two groups of experienced researchers to formalize functions with different complexities. One group formalizes functions manually and the other group formalizes functions using our prototype tool. The participants in both groups will be required to record the time cost for each function. This experiment can help us validate whether the approach is efficient in supporting experts and clarify its effectiveness in formalizing functions with different complexities.

Moreover, Individual patterns need to be expanded and more patterns need to be created to handle more complex situations. Besides, the self-learning mechanism for updating the pattern-based knowledge base is also one of our future researches. Besides, the *specification pattern knowledge* can only deal with the formalizations of bottom-level functions. To provide more intelligent guidance, we will cooperate with domain experts to design specification patterns for formalizing domain-specific functions.

In addition to the above factor, the correctness of the pattern knowledge is also important to the performance of our approach. We plan to use the following three methods for tackling this problem. First, since the pattern knowledge is represented in FSM, we can visualize the knowledge using the graphical components of FSM. This facilitates the understanding and inspection of the knowledge. Besides, model checking is a mature technique for verifying FSM models and several tools have been implemented. We can formally verify the pattern knowledge using these tools. Second, since the pattern knowledge represented in FSM can be regarded as a formal specification, formal specification inspection technique would be suitable for improving the quality of the pattern knowledge. Third, testing of the prototype tool can also help us explore the correctness of the pattern knowledge. We plan to invite students and industry people to use our tool and record the bugs or incorrect behaviors. Based on these pieces of information, the errors in the pattern knowledge can be found and removed.

At present, the element set for each specification pattern is identified based on our own understanding and experience. Whether the design of these elements is reasonable enough needs to be evaluated by experiments. Given a set of tree representations of various specification patterns, a group of developers from industry will be invited to clarify requirements using the proposed approach. Their feedback and the clarified requirements will be analyzed to improve the quality of element sets and the corresponding element definitions.

Besides, the current method for type declaration has only applied on several small systems and its intelligence still needs to be improved. We plan to carry out this method to large-scale industrial software systems to explore its problems in real application. We also plan to analyze much more numbers of industrial formal specifications to add more knowledge for supporting the method. This will probably make the method more intelligent.

Another future research is to expand this approach for formal specification evolution where formal specification gradually becomes more mature and complete.

9.2.2 Future research on the supporting tool

Sometimes, the informal guidance given by the tool is not easy to understand. Due to the inherent complexity of data structures in empirical systems, the generated guidance often involves a large number of objects and sophisticated relations. One solution is to adopt simple formal expressions in describing part of the guidance since they can be more comprehensible than their informal counter-part. Experiments need to be held to investigate this feasibility.

Another important work is to enhance the fault tolerance of the prototype tool to make it sufficiently mature to be applied by practitioners from the software industry. In our experiments, we have prepared an instruction for each participant which explicitly describes the steps for accomplishing the required formal specification with our tool and emphasized to the students to exactly follow the instruction. The reason is that if they interact with the tool in a way that is different from the standard procedure, the tool may crash or response them with an unexpected message. On the other hand, we could not expect the practitioners to always use the tool according to the instruction step by step and remain happy when the tool destroys all the previous clarified requirements just because of a wrong tack. Therefore, the ability of the tool for handling exceptional operations need to be enhanced to reach the goal of our research — facilitating the practitioners to build formal specifications.

We are also interested in developing techniques for automatically adding new knowledge to make the tool support more intelligent, as well as the techniques for supporting type and variable declarations and architecture design to support the whole process of formal specification construction.

References

[1] W. Wong, V. Debroy, A. Surampudi, H. Kim, and M. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, 2010, pp. 14–22.

[2] "Northeast blackout of 2003," http://en.wikipedia.org/wiki/Northeast_blackout_of_2003.

[3] "Program glitch led to Russian mars probe failure." [Online]. Available: http://en.rian.ru/russia/20120131/171039774.html

[4] "Natwest customers still unable to see bank balances on sixth day of glitch," *The Daily Telegrph*, 2012.

[5] "Coverity development testing solution for medical device software validation." [Online]. Available: http://www.coverity.com/library/pdf/governance-risk-and-compliance.pdf

[6] C.-F. Fan, S. Yih, W.-H. Tseng, and W.-C. Chen, "Empirical analysis of software-induced failure events in the nuclear industry," *Safety Science*, vol. 57, no. 0, pp. 118 – 128, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925753513000404

[7] N. G. Leveson, "The role of software in spacecraft accidents," *AIAA Journal of Spacecraft and Rockets*, vol. 41, pp. 564–575, 2004.

[8] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill Higher Education, 2001.

[9] "Software engineering," in *NATO Scientific Affairs Division*, 1968.

[10] O. Minor and J. Armarego, "Requirements engineering: a close look at industry needs and a model curricula," *Australasian Journal of Information Systems*, vol. 13, no. 1, 2007.

[11] "IEEE guide to software requirements specifications," *IEEE Std 830-1984*, 1984.

[12] B. Meyer, *Eiffel: The Language.* Prentice Hall Object-Oriented Series, 1991.

[13] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll, "An

overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[14] C. Snook and R. Harrison, "Practitioners' views on the use of formal methods: an industrial survey by structured interview," *Information and Software Technology*, vol. 43, no. 4, pp. 275–283, March 2001.

[15] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa, *Rigorous Software Development: An Introduction to Program Verification.* Springer, 2011.

[16] I. Morrey, J. Siddiqi, R. Hibberd, and G. Buckberry, "A toolset to support the construction and animation of formal specifications," *J. Syst. Softw.*, vol. 41, no. 3, pp. 147–160, June 1998. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(97)10016-4

[17] A. Mashkoor and J.-P. Jacquot, "Stepwise validation of formal specifications," in *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, ser. APSEC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 57–64. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2011.48

[18] S. Liu, "A simulation approach to verification and validation of formal specifications," in *Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, ser. CW '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 0113–. [Online]. Available: http://dl.acm.org/citation.cfm?id=794192.794807

[19] S. Liu, A. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "SOFL: a formal engineering methodology for industrial applications," *Software Engineering, IEEE Transactions on*, vol. 24, no. 1, pp. 24 –45, jan 1998.

[20] C.Jones, *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[21] J.M.Spivey, *The Z Notation: A Reference Manual.* Prentice Hall International (UK) Ltd, 1998.

[22] J-R.Abrial, *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[23] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 4–14, January 1998.

[24] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura, "Applying sofl to specify a railway crossing controller for industry," in *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, ser. WIFT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 16–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832314.837504

[25] J. raymond Abrial, "Formal methods : theory becoming practice," *Journal of Universal Computer Science*, pp. 619–628, 2007.

[26] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 19:1–19:36, Oct. 2009.

[27] J. Almeida, *An Overview of Formal Methods Tools and Techniques.* Springer, 2011.

[28] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: practice and experience," *ACM Computing Surveys*, vol. 41, no. 4, 2009.

[29] D. L. Parnas, "Really Rethinking Formal Methods," *Computer*, vol. 43, no. 1, pp. 28–34, 2010.

[30] S. Sahara, "An experience of applying formal method on a large business Application (in Japanese)," in *Proceedings of 2004 Symposium of Science and Technology on System Verification.* Osaka, Japan: National Institute of Advanced Industrial Science and Technology (AIST), Feb. 4-6 2004, pp. 93–100.

[31] T. Kurita, Y. Nakatsugawa, and Y. Ohta, "Applying formal specification method in the development of an embedded mobile FeliCa IC chip (in Japanese)," in *Proceedings of the 2005 Software Symposium*, Japan, June 2005, pp. 73–80.

[32] D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Software*, vol. 43, no. 1, pp. 28–34, 2010.

[33] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, "Why are formal methods not used more widely?" in *Fourth NASA Langley Formal Methods Workshop*, C. M. Holloway and K. J. Hayhurst, Eds., no. 3356, Hampton, Viginia, 1997, pp. 1–12.

[34] S. Liu, K. Takahashi, T. Hayashi, and T. Nakayama, "Teaching formal methods in the context of software engineering," *In SIGCSE Bulletin*, vol. 41, no. 2, pp. 17–23, 2009.

[35] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2nd Edition)*

(Addison-Wesley Object Technology Series). Addison-Wesley Professional, 2005.

[36] S. Vadera and F. Meziane, "From English to formal specifications," *The Computer Journal*, vol. 37, no. 9, pp. 753–763, 1994. [Online]. Available: http://comjnl.oxfordjournals.org/content/37/9/753.abstract

[37] X. Wang, S. Liu, and H. Miao, "A pattern system to support refining informal ideas into formal expressions," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, J. Dong and H. Zhu, Eds. Springer Berlin / Heidelberg, 2010, vol. 6447, pp. 662–677.

[38] X. Wang, S. Liu, and H. Miao, "A pattern-based approach to formal specification construction," in *Software Engineering, Business Continuity, and Education*, ser. Communications in Computer and Information Science, T.-h. Kim, H. Adeli, H.-k. Kim, H.-j. Kang, K. Kim, A. Kiumi, and B.-H. Kang, Eds. Springer Berlin Heidelberg, 2011, vol. 257, pp. 159–168. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27207-3_16

[39] X. Wang and S. Liu, "Computer-aided formalization of requirements based on patterns," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 2, 2014.

[40] X. Wang and S. Liu, "Guided requirements clarification for automatic formalization," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, 2013, pp. 348–355.

[41] X. Wang and S. Liu, "An approach to representing and utilizing specification pattern knowledge for computer-aided formalization of requirements," in *Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference on*, 2013, pp. 489–496.

[42] X. Wang and S. Liu, "Development of a supporting tool for formalizing software requirements," in *Structured Object-Oriented Formal Language and Method*, ser. Lecture Notes in Computer Science, S. Liu, Ed. Springer Berlin Heidelberg, 2013, vol. 7787, pp. 56–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39277-1_5

[43] L. Semmens, R. B. France, and T. W. G. Docker, "Integrated structured analysis and formal specification techniques." *Comput. J.*, vol. 35, no. 6, pp. 600–610, 1992.

[44] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and formal requirements specification languages: Bridging the gap," *IEEE Trans. Software Eng.*, vol. 17, no. 5, pp. 454–466, 1991.

[45] S. Liu, *Formal Engineering for Industrial Software Development.* Springer, 2004.

[46] S. Liu, "Integrating top-down and scenario-based methods for constructing software specifications," *Inf. Softw. Technol.*, vol. 51, no. 11, pp. 1565–1572, Nov. 2009.

[47] S. Stepney, F. Polack, and I. Toyn, "An outline pattern language for Z: five illustrations and two tables," in *ZB2003: Third International Conference of B and Z Users, Turku, Finland*, ser. LNCS, D. Bert, J. P. Bowen, S. King, and M. Walden, Eds., vol. 2651. Springer, 2003, pp. 2–19.

[48] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 31–40.

[49] J. Ding, L. Mo, and X. He, "An approach for specification construction using property-preserving refinement patterns," in *SAC*, 2008, pp. 797–803.

[50] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 372–381. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062526

[51] J. C. C. Matthew B. Dwyer, George S. Avrunin, "Pattern in property specifications for finite-state verification," in *21th International Conference on Software Engineering.* New York: ACM, 1999, pp. 411–420.

[52] J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 844–859, 2008.

[53] D. Manolescu, W. Kozaczynski, A. Miller, and J. Hogg, "The growing divide in the patterns world," *Software, IEEE*, vol. 24, no. 4, pp. 61–67, 2007.

[54] J. Ackermann and K. Turowski, "A library of OCL specification patterns for behavioral specification of software components," in *Proceedings of the 18th international conference on Advanced Information Systems Engineering*, ser. CAiSE'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 255–269. [Online]. Available: http://dx.doi.org/10.1007/11767138_18

[55] K. Miriyala and M. Harandi, “Automatic derivation of formal software specifications from informal descriptions,” *Software Engineering, IEEE Transactions on*, vol. 17, no. 10, pp. 1126 –1142, oct 1991.

[56] M. Elbendak, P. Vickers, and B. N. Rossiter, “Parsed use case descriptions as a basis for object-oriented class model generation,” *Journal of Systems and Software*, vol. 84, no. 7, pp. 1209–1223, 2011.

[57] W. McUmber and B. Cheng, “A general framework for formalizing UML with formal languages,” in *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, may 2001, pp. 433 – 442.

[58] C. Plock, B. Goldberg, and L. Zuck, “From requirements to specifications,” in *Engineering of Computer-Based Systems, 2005. ECBS ’05. 12th IEEE International Conference and Workshops on the*, april 2005, pp. 183 – 190.

[59] C. Snook and M. Butler, “U2B - a tool for translating UML-B models into B,” in *UML-B Specification for Proven Embedded Systems Design*, J. Mermet, Ed. Springer, April 2004, no. DSSE-TR-2003-3, chapter: 6. [Online]. Available: http://eprints.soton.ac.uk/260676/

[60] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, “An overview of RoZ: A tool for integrating UML and Z specifications,” in *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, ser. CAiSE ’00. London, UK, UK: Springer-Verlag, 2000, pp. 417–430. [Online]. Available: http://dl.acm.org/citation.cfm?id=646088.680051

[61] Cabral and A. Gustavo, Sampaio, “Automated formal specification generation and refinement from requirement documents,” *Journal of the Brazilian Computer Society*, vol. 14, no. 1, pp. 87–106, 2008.

[62] B. Macias and S. G. Pulman, “A method for controlling the production of specifications in natural language,” *The Computer Journal*, vol. 38, no. 4, pp. 310–318, 1995.

[63] L. Jin and H. Zhu, “Automatic generation of formal specification from requirements definition,” in *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, nov 1997, pp. 243 –251.

[64] D. Ilic, “Deriving formal specifications from informal requirements,” *Computer Software and Applications Conference, Annual International*, vol. 1, pp. 145 –152, 2007.

[65] C. Dongmo and J. A. van der Poll, "Use case maps as an aid in the construction of a formal specification." in *MSVVEIS*, D. Moldt, J. C. Augusto, and U. Ultes-Nitsche, Eds. INSTICC PRESS, 2009, pp. 3–13. [Online]. Available: http://dblp.uni-trier.de/db/conf/vveis/msvveis2009.html#DongmoP09

[66] H. Miao, L. Liu, C. Yu, J. Ming, and L. LI, "Z user studio: An integrated support tool for Z specifications," in *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, ser. APSEC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 437–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872020.872429

[67] "Rodin platform and plug-in installation." [Online]. Available: http://www.event-b.org/install.html

[68] [Online]. Available: http://www.vdmtools.jp/en/

[69] J.-R. Abrial, *The B-book: assigning programs to meanings.* New York, NY, USA: Cambridge University Press, 1996.

[70] S. Konrad and B. H. C. Cheng, "Facilitating the construction of specification pattern-based properties," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, Washington, DC, USA, 2005, pp. 329–338.

[71] O. Mondragon, A. Q. Gates, and F. Kassem, "Automated support for property specification based on patterns," in *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), Hotel Sofitel, San Francisco Bay, CA, USA, July 1-3, 2003*, 2003, pp. 174–181.

[72] G. Kotonya and I. Sommerville, *Requirements Engineering - Processes and Techniques.* John Wiley & Sons, 1998. [Online]. Available: http://www.comp.lancs.ac.uk/computing/resources/re/

[73] E. Knauss, D. Damian, G. Poo-Caamano, and J. Cleland-Huang, "Detecting and classifying patterns of requirements clarifications," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*, sept. 2012, pp. 251 –260.

[74] J. Siddiqi, I. Morrey, R. Hibberd, and G. Buckberry, "Towards a system for the construction, clarification, discovery and formalisation of requirements," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, apr 1994, pp. 230 –238.

[75] L. Jin and H. Zhu, “Automatic generation of formal specification from requirements definition,” in *Proceedings of the 1st International Conference on Formal Engineering Methods*, ser. ICFEM ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 243–. [Online]. Available: http://dl.acm.org/citation.cfm?id=523981.852164

[76] D. Ilic, “Deriving formal specifications from informal requirements,” *2012 IEEE 36th Annual Computer Software and Applications Conference*, vol. 1, pp. 145–152, 2007.

[77] L. P. Gorm, B. Nick, F. Miguel, F. John, L. Kenneth, and V. Marcel, “The overture initiative integrating tools for VDM,” *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 1–6, 2010.

[78] J. Chen and B. Durnota, “Type checking classes in object-Z to promote quality of specifications,” 1994.

[79] http://spivey.oriel.ox.ac.uk/mike/fuzz/.

[80] F. John, L. P. Gorm, and S. Shin, “VDMTools: advances in support for formal modeling in VDM,” *SIGPLAN Notices*, vol. 43, no. 2, pp. 3–11, 2008.

[81] J.-R. Abrial, *Modelling in Event-B: System and Software Design.* Cambridge University Press, 2010.

[82] S. Owre and S. Natarajan, “A brief overview of PVS,” in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5170, pp. 22–27.

[83] J. Rushby, S. Owre, and N. Shankar, “Subtypes for specifications: predicate subtyping in PVS,” *Software Engineering, IEEE Transactions on*, vol. 24, no. 9, pp. 709 –720, sep 1998.

[84] X. Tan, Y. Wang, and C. Ngolah, “A novel type checker for software system specifications in RTPA,” in *Electrical and Computer Engineering, 2004. Canadian Conference on*, vol. 3, may 2004, pp. 1549 – 1552 Vol.3.

[85] M. Xavier, A. Cavalcanti, and A. Sampaio, “Type checking circus specifications,” *Electron. Notes Theor. Comput. Sci.*, vol. 195, pp. 75–93, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2007.08.027

[86] C. Snook and M. Butler, “UML-B: Formal modeling and design aided by UML,” *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 92–122, Jan. 2006. [Online]. Available: http:

//doi.acm.org/10.1145/1125808.1125811

[87] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer Berlin / Heidelberg, 2007, vol. 4735, pp. 436–450.

[88] F.Robert and S. Ghosh, "A UML-based pattern specification technique," *IEEE Transactions on Software Engineering*, vol. 30, no. 3, pp. 193–206, 2004.

[89] S. Neelam and H. Jason, "Responsibilities and rewards : Specifying design patterns," in *26th International Conference on Software Engineering*, 2004, pp. 666–675.

[90] R. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *Software Engineering, IEEE Transactions on*, vol. 30, no. 3, pp. 193 – 206, march 2004.

[91] N. Soundarajan and J. Hallstrom, "Responsibilities and rewards: specifying design patterns," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, may 2004, pp. 666 – 675.

[92] J. M. Smith and D. Stotts, "Elemental design patterns: A formal semantics for composition of oo software architecture," in *In Proc. of 27th Annual IEEE/NASA Software Engineering Workshop*, 2002, pp. 183–190.

[93] S. Rinderle-Ma, M. Reichert, and B. Weber, "On the formal semantics of change patterns in process-aware information systems," in *Proceedings of the 27th International Conference on Conceptual Modeling*, ser. ER '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 279–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87877-3_21

[94] G. Wei, X. Sheng, and X. R. Zuo, "Formal semantics for component assembly pattern of software architecture," in *Industrial Engineering and Engineering Management, 2008. IEEM 2008. IEEE International Conference on*, 2008, pp. 2186–2190.

[95] M. S. Christopher Alexander, Sara Ishikawa, *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[96] R. J. Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[97] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*, 3rd ed. Prentice Hall, Oct. 2004. [Online]. Available: http://www.worldcat.org/isbn/0131489062

[98] X. Wang and S. Liu, "An approach to declaring data types for formal specifications," in *Structured Object-Oriented Formal Language and Method*, ser. Lecture Notes in Computer Science, S. Liu, Ed. Springer Berlin Heidelberg, 2014.

[99] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, 2002.

[100] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[101] S. Crozat, O. Hu, and P. Trigano, "EMPI: A questionnaire based method for the evaluation of multimedia interactive pedagogical software." in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 1999, pp. 1437–1443. [Online]. Available: http://dblp.uni-trier.de/db/conf/pdpta/pdpta1999-3.html#CrozatHT99