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

WANG, Xi / 王, 晳

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
9
(開始ページ / Start Page)
1
(終了ページ / End Page)
8
(発行年 / Year)
2014-03

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

Xi Wang
Department of Computer Science, Hosei University, Japan

Abstract—**Formalizing requirements in formal specifications usually requires high skills for abstraction and experience in using the formal notation in which the specification is written. This poses a challenge to many practitioners who have to deal with realistic systems within the required schedule and budget. To handle this challenge, this thesis describes a *pattern-based approach* to facilitate 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 to formalize. 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. We also describe a prototype tool that supports the pattern-based approach.**

I. Introduction

Well-written requirements specifications are the key to successful software projects. 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 serve as benchmarks for evaluating and improving the quality of the produced software. In the early days, natural language and graphical notation are used to write requirements specification since they are easy to use and understand. However, the inevitable ambiguities in informal languages would probably lead to specifications that are 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.

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 and serves as prerequisite for verifying the correctness of the implementation alternatives using formal proof or specification-based testing. In spite of the statistic data that shows the improvement of software quality by using formal specification techniques and successful stories reported in the latest survey on industrial use of formal specification, applications of specification to real projects in industrial are still rare. The major reason stems from the nature complexity of formal notations. Formalizing informal requirements into formal specifications requires sophisticated mathematical skills, abstraction ability and sufficient experience in using formal notations. This poses a challenge to practitioners who hardly receive any systematic training in computer science but have to face pressures to produce qualified software systems within limited time and budget. Even if they manage to understand formal notations through long-term practice, formalizing complex functions is still error-prone and costly.

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 expressions 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.

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 predefined 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. 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 developer in writing formal specifications; they are "understood" only by the computer but transparent to the developer. To enable automatic application, we give the formal definition 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.

In addition to the design and definition of the *specification 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 to facilitate understanding of the produced guidance and utilization of the pattern knowledge. We also describe a prototype tool that implements the pattern-based approach. 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.

II. Related Work

Many methods have been proposed for supporting requirements formalization. They can be generally divided into two kinds. The first kind focuses on the construction of formal specifications. In (1), the authors describe some example approaches to integrating structured methods of software development with formal notations. In (2), authors uses the structured analysis (SA) model of a system to guide the analyst's understanding of the system and the development of the VDM specifications. S. Liu (3) proposes an approach to constructing software specifications by integrating top-down and scenario-based methods. These methods either do not provide any guidance for formal specification construction or provide guidelines on an abstract level. There are also many methods for supporting formal specification construction using patterns. Stepney *et al.* describe a pattern language for using notation Z in computer system engineering (4). Lars Grunske presented a specification pattern system of common probabilistic properties for probabilistic verification (5). Konrad *et al.* (6) create real-time specification patterns in terms of three commonly used real-time temporal logics based on an analysis of timing-based requirements of several industrial embedded system applications and offer a structured English grammar to facilitate the understanding of the meaning of a specification. 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. By contrast, our pattern-based approach is able to provide specific comprehensible guidance by automatic application of the pattern system. Developers only need to follow the guidance and the rest of the tasks will be handled by the tool.

The second kind emphasizes the solution to the conversion of informal requirements to formal specifications. 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 (7). Cory Plock *et al.* show how to transform LSC (Live Sequence Charts) specifications with concurrency to timed automata (8). Sunil Vadera *et al.* propose an interactive approach for producing formal specifications from English specifications (9). Several tools have also been proposed for automatic transformation, such as U2B (10). But such transformation is conducted based on certain pre-defined syntactic rule without considering the real meaning of the models. Due to the inherent difficulty in NLP technology, there is no effective tool-support in constructing formal specifications on the semantic level. The introduction of pattern seems to offer a solution. 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.

Several kinds of tools have been developed for supporting requirements formalization. Z User Studio is developed as an integrated Z support tool (11). U2B translator (10) converts UML Class diagrams, including attached state charts, into the B notation (12). SPIDER (13) derives and instantiates system properties in terms of their natural language representations. Kanth Miriyala *et al.* describe an interactive system called SPECIFIER for deriving formal specifications of data types and programs from their informal descriptions (14). These tools is either unable to work on the semantic level or incapable of dealing with data-intensive systems without manual efforts on formal notations. By contrast, our tool supports the requirements formalization process on the semantic level and the developers can focus function design without the need of considering the formal notation details.

III. An Overview on the Pattern-Based Approach to Requirements Formalization

The outline of our pattern-based approach is given in Figure ?? 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*. The former specifies each attribute of the requirements to obtain sufficiently detailed understanding while the latter accomplishes the declaration of all the necessary data types for formalizing the requirements. Instead of being performed independently, these two activities are alternatively 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

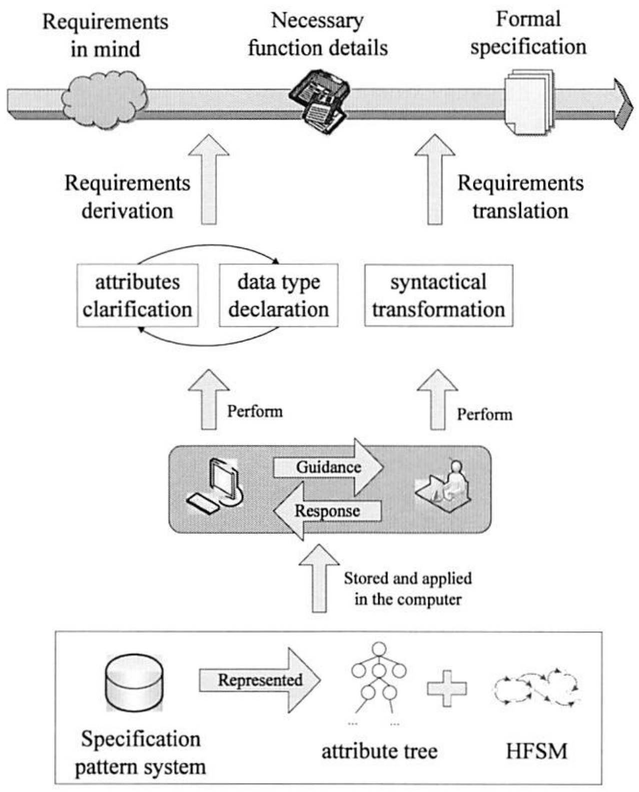


Fig. 1. The outline of the pattern-based approach

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 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 in 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 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 other parts of *derivation knowledge* and all *transformation knowledge*, since it is easy to be manipulated by machines and maintained by several mature supporting tools.

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 *specification pattern system* will be first introduced and the representation of the included pattern knowledge is then described. Based on the pattern knowledge, we will show how to apply it for supporting the two stages during requirements formalization.

IV. Specification Pattern System

Specification pattern system is composed of a set of inter-related *specification patterns*. We will first present the concept of *specification pattern* and then shows how these patterns are organized in the hierarchy of the pattern system.

A. Specification pattern

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. Specifically, our *specification pattern* is 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 functions 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

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 to 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
- $\Phi : \Phi_E \cup \Phi_R$ denotes the set of clarification rules for guiding the assignment of the elements in E where
  - $\Phi_E : E \longrightarrow E$ 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)
    - $\forall_{e \rightarrow e' \in \Phi_E} \cdot e \neq e'$ ($e \rightarrow e'$ 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 (R, R_0, γ) where
    - $R : \mathcal{P}(PR) \longrightarrow \mathcal{P}(PR)$ denotes the set of rules in the repository where each rule determines the satisfactory of a set of constraints based on already satisfied constraints and $\forall_{PR_i \rightarrow PR'_i \in R} \cdot PR_i \cap PR'_i = \varnothing$
    - $R_0 \subset R$ is the first set of candidate rules to be applied
    - $\gamma : R \rightarrow \mathcal{P}(R)$ determines the sequence for applying the rules in R where $\gamma(r)$ indicates the candidate rules for further clarifying e after rule r is applied
    - $\forall_{RS_i \in ran(\gamma)} \cdot \forall_{PR_m \rightarrow PR'_m, PR_n \rightarrow PR'_n \in RS_i} \cdot \forall_{pr \in PR_m} \cdot pr \Rightarrow \exists_{pr' \in PR_n} \cdot \neg pr'$ (for each $r \in R$, only one of the candidate rules in $\gamma(r)$ will be activated when formalizing a function using p)
- $\Psi : \mathcal{P}(PR) \longrightarrow string$ 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 *name* 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 four elements in E by defining Φ_E and Φ_R. Mapping Ψ mathematically defines item *solution*.

B. The hierarchy in the pattern system

Due to the inherent complexity of software, the number of patterns will be so large that the selection process becomes a hard task for users and the management would be difficult. To this end, 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. Figure 2 shows the hierarchy where rightmost items represent patterns and others represent categories. Root "Pattern system" owns two sub-items, which indicates that patterns are divided into two categories: one for describing unit functions denoted as UF and the other for depicting compound functions denoted as CF. Their sub-categories are further classified into more specific sub-categories or patterns. For example, category UF is divided into three sub-categories: *Relation* patterns for formalizing the descriptions of relationships between objects, *Retrieval* patterns for generation formal expressions representing system variables and *Recreation* patterns for formal description of changes on the state of the system.

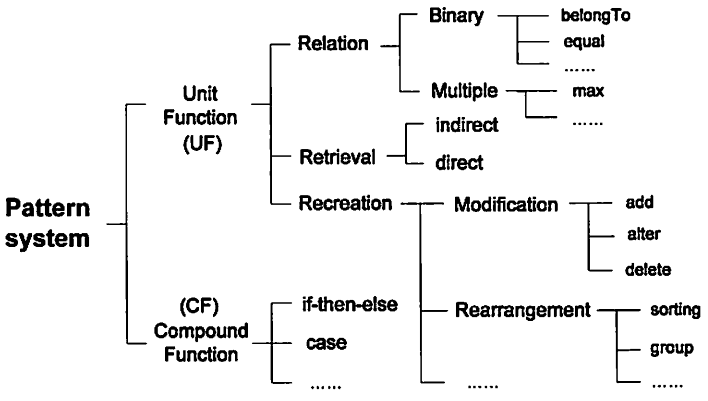


Fig. 2. Pattern categorization

V. 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 need to be formally defined by custom data types. Instead of performing sequentially, these two tasks are usually alternatively 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.

A. From informal requirements to formal specifications

Well-defined pattern structure 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.

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. Human intelligence is needed in this step 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.

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) requirements 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.

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.

B. Data type declaration

As the complexity of software rises, data type declaration becomes more difficult to manage and more likely to result in defected data types. 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.*

The proposed approach regards data type declaration as an evolution process along with the writing of formal expressions based on function patterns. This evolution process starts with a modulized formal specification and terminates when the detailed behavior of each module is precisely given. 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 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.

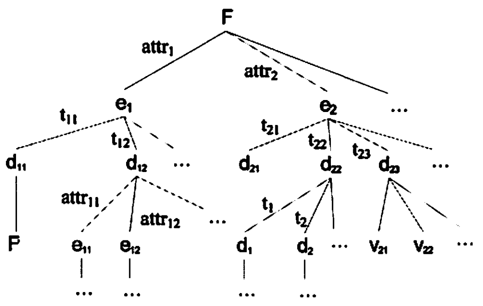


Fig. 3. The structure of attribute tree

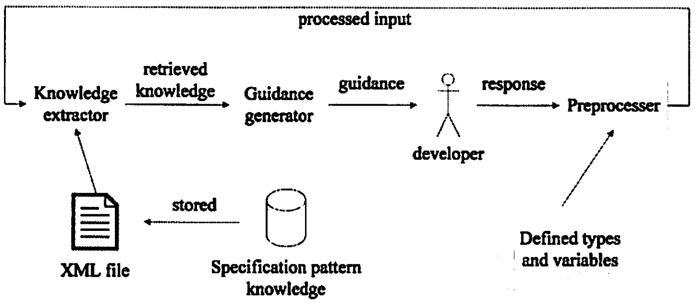


Fig. 4. The design of the tool for supporting requirements formalization

When the formalization process terminates with a formal expression, a refined data context is obtained. Finally, *expression update* is carried out where all the formal expressions that are inconsistent with the refined data context are updated.

VI. Representation of the Pattern Knowledge

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. 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 people-oriented knowledge representation, attribute tree and HFSM are chosen to describe the two kinds of knowledge respectively.

A. Attribute tree

Figure 3 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 $e_1, e_2, \ldots$ 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 $d_{i1}, d_{i2}, \ldots$ 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}.

B. HFSM

HFSM models system behaviors using a set of FSMs organized in a hierarchy. The definitions of FSM and HFSM are first given before explaining how to represent pattern knowledge.

Definition 2: 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.

Definition 3: 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).

The HFSM representing the target 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 in pattern system application. 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 the detailed process of pattern selection. For step 2, requirements clarification and formal expression generation are modeled by a set of lower-level FSM each describing the application of one of the patterns.

VII. Prototype Tool for Supporting the Pattern-Based Approach

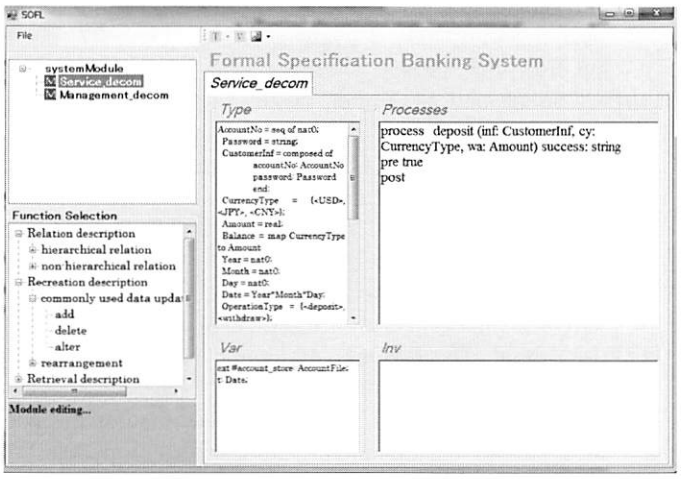
The main goal of our pattern-based approach is to support computer-aided formalization of software requirements. To validate the approach and demonstrate its efficiency, we implement it into a prototype tool that implements the approach. It interacts with the developers to derive necessary function details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

Figure 4 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*

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 is 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.

Figure 5 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;
```

Fig. 5. 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. 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.

For each function to be formalized, a pattern should be selected from the tree structure on the bottom left of the main frame. Once the selection decision is made, a new frame will be poped up as the medium to derive necessary function details of the intended basic function. 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.

TABLE I
THE RESULT OF THE FIRST EXPERIMENT

Software system	Number of processes	Number of applied patterns	Number of errors
H	53	10	13
B	49	9	11
E	50	9	10
S	53	11	13
L	55	12	10
O	60	12	11

VIII. Experiment

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

In the first experiment, we use the supporting tool to formalize the functions of several typical software systems including 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). Table I shows the result of the experiment. 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.

In the second experiment, we invite 76 students and divided them into two groups. Each student in group 1 is asked to manually write the formal specification of a banking system and the students in group 2 formalize the behavior of the banking system by using our prototype tool. The result of the experiment is organized in Table II. As can be seen from the table, the tool help formalize requirements more efficiently and enhance the quality of the resultant formal expressions.

IX. Conclusion and Future Work

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. It includes a set of patterns categorizes these patterns in a hierarchy according to the functions they are used to formalize. A

TABLE II
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

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. Furthermore, a prototype tool that implements the proposed pattern-based approach is developed and described.

Sometimes, the informal guidance given by the tool is not easy to understand. 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. Moreover, the correctness of the pattern knowledge is also important to the performance of our approach. We will carry out formal verification, inspection and testing technologies to check the correctness in our future research work.

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] 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.

[2] 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.

[3] 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.

[4] S. Stepney, F. Polack, and I. Toyn, "An outline pattern language for Z: five illustrations and two tables," in *Third International Conference of B and Z Users pages =*.

[5] L. Grunske, "Specification Patterns For Probabilistic Quality Properties," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 31–40.

[6] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 372–381.

[7] 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.

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

[9] S. Vadera and F. Meziane, "From english to formal specifications," *The Computer Journal*, vol. 37, no. 9, pp. 753–763, 1994.

[10] C. Snook and M. Butler, "U2b - a tool for translating uml-b models into b," in *UML-B Specification for Proven Embedded Systems Design.* Springer, April 2004, no. DSSE-TR-2003-3, chapter: 6.

[11] 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.* IEEE Computer Society, 2001.

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

[13] 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.

[14] 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.