Computer-Aided Formalization of Requirements Based on Patterns

Xi WANG†(a), Member and Shaoying LIU†, Nonmember

SUMMARY
Formalizing requirements in formal specifications is an effective way to deepen the understanding of the envisioned system and reduce ambiguities in the original requirements. However, it requires mathematical sophistication and considerable experience in using formal notations, which remains a challenge to many practitioners. To handle this challenge, this paper describes a pattern-based approach to facilitate the formalization of requirements. In this approach, a pattern system is predefined to guide requirements formalization where each pattern provides a specific solution for formalizing one kind of function into a formal expression. All of the patterns are classified and organized into a hierarchical structure 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 guidance to facilitate the developer in requirements formalization; they are “understood” only by the computer but transparent to the developer. We also describe a prototype tool that supports the approach. It adopts Hierarchical Finite State Machine (HFSM) to represent the pattern knowledge and implements an algorithm for applying it to assist requirements formalization. Two experiments on the tool are presented to demonstrate the effectiveness of the approach.

key words: formal specification, pattern system, HFSM, computer-aided

1. Introduction

Requirements formalization is a process of constructing formal specifications by clarifying relevant function details and representing the clarified requirements in formal notations. It helps deepen the understanding of the envisioned system and significantly improves the precision of the original requirements. With well-established formalism, the resultant formal specifications can be manipulated automatically and serve as a prerequisite for verifying the correctness of the implementation alternatives using formal proof or specification-based testing [1]. Unfortunately, most of the practitioners are still confronting challenges in formalizing requirements. They lack of sufficient skills and experience in identifying the function details necessary to be clarified and describing functions with appropriate mathematical statements [2]. Even if they manage to understand formal notations through long-term practice, formalizing complex functions is still error-prone and costly. Therefore, the kind of automatic support to provide for requirements formalization, as well as how to provide it, is a crucial problem to be solved in order for software industry to benefit from the formal specification technique.

Copyright © 2014 The Institute of Electronics, Information and Communication Engineers
inition 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. And a case study is presented to illustrate the method.

We also describe a prototype tool that implements the pattern-based approach. It consists of a knowledge base for storing the pattern system and the implementation of an algorithm for utilizing the stored pattern knowledge to determine the behavior of the tool. The knowledge base stores the pattern system in terms of a set of HFSMs (Hierarchical Finite State Machines) where each HFSM represents the knowledge included in an individual pattern. On the basis of the HFSM representation, the algorithm for utilizing the pattern knowledge is given. Two experiments on the prototype tool were conducted and their results demonstrate the effectiveness of the pattern-based approach.

Note that the underlying principle of the approach is language-independent. We choose SOFL (Structured Object-oriented Formal Language) to illustrate how the approach works in this paper because we are familiar with SOFL. One can refer to [1] for the details of the language.

The remainder of this article is organized as follows. Section 2 introduces the pattern system and gives the formal definition of pattern. Section 3 describes the method for guiding requirements formalization based on the pattern system and illustrates the method through a case study. Section 4 presents the prototype tool that implements the proposed approach and two experiments for evaluating the performance of the tool. Section 5 reviews the related work. Finally, Sect. 6 concludes the paper and discusses the future work.

2. Pattern System

The pattern system is composed of a set of patterns each dealing with the formalization of one kind of function on the assumption that all the necessary types and variables are already defined. 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 kind of functions they can be used to formalize. We will present pattern definition and pattern categorization respectively.

2.1 Pattern Definition

A pattern needs to tackle two tasks when formalizing a kind of function \( f \): the clarification of \( f \) and the representation of the clarified \( f \) in a formal notation. Therefore, our pattern mainly consists of two parts providing solutions for the above two tasks respectively. In the first part, \( f \) is treated as a composition of its necessary attributes. These attributes are formally defined and clarifying \( f \) is to assign values to the attributes 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 attributes. 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.

In our approach, the patterns are designed to be applied by machines; any ambiguity will impede their automatic utilization. For this reason, a formal definition for pattern is given as follows 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 a necessary attribute of \( f \)
- \( PR \) is a set of constraints on \( p \) or the elements in \( E \)
- \( expl: \{f\} \cup E \cup PR \rightarrow string \) 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
  - \( \Phi_E: E \rightarrow E \) 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 \( \Phi_E \) where \( e' \) should be specified after \( e \))
  - \( \Phi_R: E \rightarrow RPT \) defines a rule repository for each element \( e \) in \( E \) to guide the assignment of \( e \) where each repository in \( RPT \) is a triple \((R, R_0, \gamma)\) where
    * \( R: \mathcal{P}(PR) \rightarrow \mathcal{P}(PR) \) denotes the set of rules in the repository where each rule determines the satisfiability of a set of constraints based on already satisfied constraints and \( \forall_{PR \rightarrow PR_r \in R} \cdot PR_i \cap PR'_j = \emptyset \)
    * \( R_0 \subset R \) 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
    * \( \forall_{RS_0 \in ran(\gamma)} \cdot \forall_{PR_r \rightarrow PR_{r'}} \cdot PR_i \in RS_0 \cdot PR_{r'} \in PR_{r'} \cdot pr \Rightarrow \exists_{pr' \in PR_{r'}} \cdot \neg pr' \) (for each \( r \in R \), only one of the candidate rules in \( \gamma(r) \) will be activated when formalizing a function using \( p \))
- \( \Psi: \mathcal{P}(PR) \rightarrow string \) denotes the set of transformation rules for generating the formal representation of \( f \) according to the values assigned to the elements where
In the definition, an element denotes an attribute of \( f \) and item \( E \) consists of all the necessary attributes to be clarified to formalize \( f \). Each element is formally defined with one of the element types given in Table 1 according to the nature of the corresponding attribute. 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. The first six types after \( nil \) are atomic types for defining elements that can be clarified without further decomposition. And the rest types are structured types for defining elements that need to be clarified by low-level elements. For each element \( e \) of atomic type \( t \), definition format is \( e : t \). Elements of different structured types use different formats. For an element \( e \) of composite type with fields \( f_1, \ldots, f_n \), its definition format is: \( e : f_1 \times \cdots \times f_n \). For an element \( e \) defined as a set of elements \( e' \), its definition format is \( e : \{ e' \} \). 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 \( p \)
- \( e : p(v_1, \ldots, v_n) \): let the item \( E \) of the pattern \( p \) be \( \{ e_1, \ldots, e_m \} \) \((m \geq n)\), element \( e \) will be assigned with a function formalized by applying pattern \( p \) with each \( e_i \) specified as \( v_i \) where \( \forall e, e_i \in E : e_i = \Phi e_e(v) \)
- \( e : c(v_1, \ldots, v_n) \): element \( e \) ranges over a set of formalized functions \( \{ f_1(v_1, \ldots, v_n), \ldots, f_m(v_1, \ldots, v_n) \} \) where category \( c \) consists of \( m \) patterns \( \{ p_1, \ldots, p_m \} \) (The concept of category will be introduced in the next section).

There are three elements in the pattern \( sort \), each denoting a necessary attribute for composing a sort function (the attributes that these elements stand for will be presented when explaining the \( expl \) item of the definition). Elements \( obj \) and \( result \) are both defined as \( varValue \) 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.

Item PR includes three kinds of constraints. The first kind is the propositions on the pattern \( p \), such as the constraint \( \text{Reuse in the PR} \) item of the pattern sort which means that “\( p \) is applied to describe sub-functions for the application of other patterns”. The second kind is definition constraints, i.e., the constraints on element definition. For example, element ruleType is initially defined as a choice type with four candidate items. Constraint \( \text{ruleType : \{et, gr\}} \) refines this 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 third kind is value constraints, i.e., the constraints on element value, such as \( \text{ruleType = et and dataType(objs) = char} \) meaning that the value assigned to element objs is a character. 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.

Item expl converts three kinds of objects into corresponding informal expressions for forming comprehensible guidance. The first object is \( f \) and \( \text{expl}(f) \) 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. For example, the first maplet in the expl item of the pattern sort provides an explanation on sort function. The second kind is the elements included in \( E \). For each element \( e \), \( \text{expl}(e) \) indicates the attribute of \( f \) that \( e \) stands for. In the expl item of the pattern sort, for example, maplets 2, 3, 4 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. The third kind is the definition constraints and value constraints in PR. For each definition constraint or value constraint \( pr \) on element \( e \), \( \text{expl}(pr) \) indicates the informal guidance that requires for assigning \( e \) with a value satisfying \( pr \). Maplet 5 in the expl item of the pattern sort gives an example. 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.

Item \( \Phi \) 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 \( \Phi \) to guide the assignments of the elements in \( E \). These rules are divided into two groups. The first group \( \Phi_E \) reveals the order for specifying the elements in \( E \) where each rule \( e \rightarrow e' \) means that element \( e \) should be assigned before \( e' \). For example, in the item \( \Phi_E \) of the pattern sort, rule result \( \rightarrow \) rule indicates that element rule is required to be specified after result is specified.

The second group \( \Phi_R \) provides a rule repository for guiding the clarification of each element in \( E \) step by step. Each rule repository \( (R, R_0, \gamma) \) owns a rule set \( R \) and determines the sequence for applying the rules in \( R \). A rule in \( R \) infers new constraint 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. The sequence for applying the rules is \( R_0, R_1, \ldots, R_n \). Each \( R_i \) denotes the \( i \)th applied set of candidate rules where each candidate rule is a rule in \( R \) 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.

Let’s take the item \( \Phi_R \) of the pattern sort as an example where the rule repositories of the three elements are given. The rule repository for \( \text{objs} \) is \( (R, \{r1\}, \gamma) \) where three example rules \( r1, r2, r3 \) in \( R1 \) are listed and \( \{r1\} \) is the first candidate rule set to be applied. Premise constraint \( \text{true} \) 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 “\( \text{objs : varValue} \)”. This constraint serves as a guideline that asks for the assignment of \( \text{objs} \) with a value of varValue type. When the response from the developer is received, candidate rule set \( \gamma(r1) \), i.e., \( \{r2, r3, \ldots\} \) will be applied by applying its activated rule. Assigning different values to \( \text{objs} \) leads to different activated rules. For example, if the given \( \text{objs} \) is a character, \( r2 \) will be activated and applied where the retrieved constraint \( \text{re(objs)} \) means that \( \text{objs} \) should be assigned again with a new value since sorting one character makes no sense (We use the keyword \( \text{re} \) in pattern definitions to represent reassignment). For each element \( e, \text{re(e)} \) means the reassignment of \( e \). But if the given \( \text{objs} \) is a set, \( r3 \) will be activated and applied where \( \emptyset \) means that \( \text{objs} \) 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 \( \gamma(fr) = \emptyset \). We skip the explanation on the rule repository of element result since it is similar to that of \( \text{objs} \). 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 \( \text{objs} \) 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 $R_3$ is determined by the item $\gamma$. For example, $\{r_8, \ldots\}$ will be applied if $r_6$ is activated when applying $R_3$ and $\{r_9, \ldots\}$ will be applied if $r_7$ is activated.

Note that the constraints derived by $r_8$ and $r_9$ in the pattern sort 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($obj$, $elems(result)$)” is used to define low-level element $gR$ in $r_9$. According to mapping 6 in the expl of the pattern sort, $gR$ 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 ($obj$, $elems(result)$). As shown in Fig. 2, three elements are included in the item $E$ of the pattern group. The $\Phi_E$ item reveals that $gobj$ 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 $gobj$ and $gresult$ are assigned as $obj$s and $elems(result)$ respectively.

Item $\Psi$ 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 $E$. In the formal expression, element names are used to denote the values assigned to the corresponding elements when applying $p$. For example, rule $tr1$ in the pattern sort 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 $\Phi$ and $\Psi$ 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.

### 2.2 Pattern Categorization

All the patterns are organized in a hierarchical structure in the pattern system by categorizing the functions they are used to formalize. Figure 3 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 and the other for depicting compound functions. Their sub-categories are further classified into more specific sub-categories or patterns. For example, $UF$ is divided into three sub-categories: Relation patterns for describing relations between objects, Retrieval patterns for obtaining the formal representations of system variables and Recreation patterns for depicting the update of system variables.

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

The final goal of the pattern system is to support the requirements formalization for general system development. But as the first step, the domain of the current pattern system described in this paper 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 be complete at present and will be updated when new patterns and categories are introduced, or existing ones are 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.

3. Requirements Formalization Based on the Pattern System

In this section, we will present the method for assisting requirements formalization based on our pattern system and simulate the formalization process using an example requirement in the case study to illustrate the method.

Given a requirement \( r_q \), its formalization based on the pattern system contains two major steps.

Step 1 Pattern selection

Appropriate patterns for formalizing \( r_q \) 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 \( r_q \) if \( r_q \) 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 \( r_q \) 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 \textit{money transfer} function for an ATM system, one cannot find a pattern specifically designed for formalizing \textit{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 \textit{Recreation} category.

Step 2 Pattern application

With a set of patterns \( \{ p_1, \ldots, p_n \} \) selected for all the sub-functions \( \{ f_1, \ldots, f_n \} \) of \( r_q \), 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

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 \( p_i \).

\[
\begin{align*}
ce &= e_0; \\
\text{// } ce \text{ denotes the element being specified }
\end{align*}
\]

while(ce is an element in \( E \))

\[
\begin{align*}
\text{// } if(ce \text{ has not been specified}) \\
\text{retrieve the rule repository } \Phi_{R}(ce) &= (R, R0, \gamma); \\
CR &= R0; \\
\text{// } CR \text{ denotes the candidate rule set being applied }
\end{align*}
\]

\[
\text{while}(CR \neq \Phi) \\
\text{apply } CR \text{ by applying its activated rule } ar; \\
\text{accordingly display guidance and receive response; } \\
CR &= \gamma(ar); \\
\}
\]

\[
\text{ce} = \Phi_{E}(ce); \\
\}
\]

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 \((PR \rightarrow str) \in \Psi\)

\[
\text{if}(\forall_{pr \in PR}, \cdot pr) \quad exp = \Psi(PR);
\]

A case study on the functions of a bank data analysis system is presented to illustrate the above two steps. This system provides analysis services for a bank data store \textit{account_store} where each authorized customer owns a banking account manipulating on different kinds of currencies. Each account includes balance and transaction history information where each transaction records an operation on the account from four aspects: the date when the operation performed, operation type, the type and amount of the currency that the operation performs on. Figure 4 shows the types and variables formally defined for describing the above bank data.

Four services are provided by the analysis system including \textit{balance analysis}, \textit{transaction analysis}, \textit{global balance analysis} and \textit{global transaction analysis}. We will take the formalization of the function \textit{transaction analysis} as an example. It extracts transactions from certain account and sorts them by dividing them into different groups by date.
Fig. 4  Types and variables declared for the bank data analysis system.

given as response to the above guidance and assigned to \( o_b j s \) (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 \( r_1 \), \( \gamma(r_1) = \{ r_2, r_3, \ldots \} \) becomes the next candidate rule set to be applied. By automatically evaluating the premise constraint for each included candidate rule, \( r_3 \) is determined as the activated one since its premise constraint can be satisfied by the given \( o_b j s \). The derived new constraint is empty and no guidance is provided. Since \( \gamma(r_3) = \emptyset \), the assignment of \( o_b j s \) is thus finished.

The \( \Phi_E \) item of the pattern sort tells that \( \Phi_E(o_b j s) = \text{result} \), i.e., element result is the next element to be specified. Again, the rule repository \( \{ r_1, [r_4], \gamma \} \) is derived to guide the assignment of \( \text{result} \). Similar to \( o_b j s \), candidate rule \( r_4 \) is first applied resulting in the constraint “result : varValue”. 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 \( \text{result} \). Candidate rule set \( \gamma(r_4) = [r_5, \ldots] \) is then applied where \( r_5 \) is activated. The derived new constraint is empty and the assignment of \( \text{result} \) terminates since \( \gamma(r_5) = \emptyset \).

Element rule is the last element to be specified. According to its rule repository \( \{ r_3, [r_6, r_7, \ldots], y \}, [r_6, r_7, \ldots] \) is first applied where \( r_7 \) 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 \( \text{rule} \). The next candidate rule set is \( \gamma(r_7) = [r_9, \ldots] \) where \( r_9 \) is activated and applied. The obtained constraint decomposes rule.content into two low-level elements \( grR \) and \( srR \). 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 \( grR \)’s definition given in the constraint, its assignment is guided by applying the pattern group with the first two elements assigned as \( o_b j s \) and \( \text{elems(result)} \) respectively (In SOFL, \( \text{elems(seq)} \) means the set of elements in sequence \( \text{seq} \)).

The application of the pattern group also follows the given two steps. Step a starts from the
assignment of element \textit{grule}, since the first two elements \textit{gobjs} and \textit{gresult} have already been assigned in \textit{gR}'s definition. According to the pattern, the rule repository of \textit{grule} is \((R^3, \{r5\}, \gamma)\) and \(r5\) is first applied resulting in constraint “\(\textit{grule.ruleType} : \{uR, iR\}\)” Based on the \textit{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 \(\textit{grule.ruleType}\). The assignment of \textit{grule} continues with the candidate rule set \(\gamma(r5) = \{r6, \ldots\}\) where \(r6\) is activated. The derived constraint allows the assignment of \textit{grule.content} by choosing from \(f1, \ldots, fn\) where each \(fi\) denotes grouping \(\textit{gobjs}\) according to the field \(fi\). In our case, each \(fi\) is instantiated as one of the fields of the transaction and the \textit{grule.content} is guided to be assigned by choosing from \textit{date}, \textit{operationType}, \textit{currencyType} and \textit{amount}. It is easy to make the selection decision according to the semantics of the example function. Item \textit{date} is selected as the response and assigned to \textit{grule.content}. Step \(a\) for applying the pattern \textit{group} is then finished since \(\gamma(r6) = \emptyset\). Then the application moves to step \(b\) with the assigned elements. By automatically evaluating the premise constraint of each rule in the \(\Psi\) item of the pattern \textit{group}, rule \textit{tr1} is activated since its premise constraint can be satisfied by the values assigned to the three elements. The formal expression suggested by \textit{tr1} is generated as the result of applying the pattern \textit{group}.

Since the above application of the pattern \textit{group} is to clarify the low-level element \textit{gR}, the generated formal expression is assigned to \textit{gR} and the algorithm goes back to the application of the pattern \textit{sort} where \textit{r9} has been activated and applied. According to \(\gamma(r9)\), candidate rule set \(\{r10, \ldots\}\) is then applied where \(r10\) is activated. The application of \(r10\) results in a constraint on the definition of the low-level element \textit{sR} which is interpreted, by the \textit{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 \textit{sort} is finished since \(\gamma(r10) = \emptyset\).

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 \textit{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 \(\Psi\) item of the pattern \textit{sort} is automatically analyzed in the context of the clarified elements to explore the activated one \(“\{pr1, \ldots, prn\} \rightarrow str”\) where constraints \(pr_1, \ldots, pr_n\) are all satisfied. String \textit{str} is then generated as the formal representation of the function \textit{transaction analysis}. Since the premise constraint of the rule \textit{tr1} can be satisfied, \textit{tr1} is applied and the corresponding formal expression is given as shown in Fig. 5.

The case study simulates the interaction process for guiding the formalization of the function \textit{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 \textit{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.

4. Prototype Tool and Experiment

4.1 Prototype Tool

We have developed a prototype tool for supporting requirements formalization based on the pattern system. It consists of two major components: a knowledge base for storing the pattern system and the implementation of an algorithm for utilizing the stored pattern knowledge to determine the behavior of the tool. We will first explain these two components in detail respectively, and then describe the tool implemented based on them.

The knowledge base stores the pattern system in terms of a set of HFSMs (Hierarchical Finite State Machines) where each HFSM represents the knowledge included in an individual pattern. The pattern categorization structure is not included in the knowledge base since it is directly given through the tool’s interface for pattern selection. An HFSM is composed of a set of FSM (Finite State Machine) models where the details of some portion of each high-level FSM is interpreted by a set of low-level FSMs.

We use HFSM, instead of the static structure of the pattern definition, to represent pattern knowledge because of the following reasons. First, the algorithm for utilizing the static structure of the pattern definition is complex and will become more and more complex as the patterns keep being updated with new knowledge. By contrast, the algorithm for analyzing HFSM is much easier and its complexity remains unchanged as the patterns become more complex. Second, each time when the pattern definition is slightly modified, the algorithm for utilizing it must be updated to deal with the newly introduced definitions and all the parties who share the knowledge have to update their tool with the new algorithm. The algorithm for utilizing HFSM, however, does not need to be modified when the pattern definition is changed, since the semantics of the basic elements involved in HFSM remain the same. Whatever modification is done on the HFSM, the original algorithm can still work on the updated knowledge, which saves much time in implementing a new one. Third, compared with other modeling languages, HFSM is precise and easier to be comprehended and manipulated by automated means. The graphical notation of HFSM demonstrates the hierarchical relation between patterns and more clearly shows the behavior of individual patterns through FSMs. Besides, many mature techniques and tools are proposed to analyze FSMs, which can be sufficiently utilized to analyze our pattern knowledge. It should be noted that the semantics of the knowledge included in a pattern definition is equivalent to that of the knowledge included in its corresponding HFSM. Their determined tool behaviors are the same. We only change the representation of the knowledge to facilitate its utilization and maintenance.

To enable automatic utilization, knowledge representation must be formally defined. The formal definitions of FSM and HFSM are given and explained through an example HFSM.

**Definition 2:** A FSM (Finite State Machine) is a 9-tuple \((Q, q_0, F, VP, I, G, \varphi, \delta, \lambda)\) where \(Q\) is a non-empty finite set of states, \(q_0 \in Q\) is the initial state, \(F \subset Q\) is the set of accept states, \(VP\) is a set of triples \((V, V', \theta)\) where \(V\) is the finite set of state variables, \(V'\) is a set of values and \(\theta : V \rightarrow V'\) indicates the value of each \(v \in V, I\) is the finite set of symbols for composing inputs and outputs, \(G\) is the finite set of guard conditions, \(\varphi : Q \rightarrow VP\) is the state function indicating the values of the involved variables on each state, \(\delta : Q \times (I \times P(G)) \rightarrow Q\) is the transition function relating two states by input and guard conditions, \(\lambda : Q \times (I \times P(G)) \rightarrow I\) is the output function determining output based on the current state and input.

**Definition 3:** HFSM (Hierarchical FSM) is a pair \((F, \sigma')\) where \(F\) is a set of FSMs and \(\sigma' = \cup_{A \in FA} \cup_{\cup_{A \in FA} A \rightarrow P(G)}\) is the hierarchical relations among FSMs in \(F\) where lower-level FSMs interpret certain portion of upper-level FSMs iff \(\exists_{A \in FA} \cdot V_{Ftrans} \cdot A_0 \notin F' (A_0\) is the top-level FSM).

Figure 6 shows an example HFSM representing the knowledge included in the pattern sort where \(\alpha\) denotes the actual input from the developer. For the sake of space, we only give parts of the top-level FSM to illustrate how a pattern is represented in the knowledge base.

The state set of the FSM is \(\{p_0, \ldots, p_n\}\) where \(p_0\) is the initial state and \(p_{13}\) is one of the accepted states. Equations attached to each state \(p_i\) reflect \(\varphi(p_i)\) meaning that the state variables will be assigned according to these equations when the FSM is transferred to \(p_i\). For example, \"\(\text{objs} = \alpha'\)\" attached to \(p_2\) indicates that state variable \(\text{_objs}\) will be assigned as \(\alpha\) when the FSM is transferred to \(p_3\).

Connecting states with arrowed lines, transitions reflect the transition function \(\delta\) and output function \(\lambda\) of the FSM. Each transition \(p_i \rightarrow p_j\) is attached with a label \(\ell(O)/o\)
where \( p_j = \delta(p_i, (i, G)) \land o = \lambda(p_i, (i, G)) \), which means that when the FSM stays on state \( p_i \), if input \( i \) is received and each \( g \in G \) is satisfied, output \( o \) will be displayed and the FSM will be transferred to state \( p_j \). Let’s take the transition \( p_0 \rightarrow p_1 \) as an example. It describes that if \( obs = null \) can be satisfied on \( p_0 \), guidance produced from the constraint \( obs : expValue \) will be displayed without input and the FSM will be transferred to \( p_1 \). Since no equation is attached to state \( p_1 \), system variables will not be updated. For transitions without the involvement of guard conditions, label \( i/o \) is used which means that the corresponding transition will be triggered when receiving \( i \) under any condition.

The example HFSM models the behavior of the pattern \( sort \) according to its application process. Paths between states \( p_1 \) and \( p_{12} \) reflect step \( a \), i.e., pattern \( sort \)’s behavior of guiding the requirement clarification process. Each path indicates the guiding process for clarifying a specific kind of function details. For example, the requirements clarification process simulated in the previous case study corresponds to the path:

\[
p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_5 \rightarrow p_6 \rightarrow p_7 \rightarrow p_8 \rightarrow p_9 \rightarrow p_{10} \rightarrow p_{11} \rightarrow \cdots \rightarrow p_{12}
\]

Transitions originated from state \( p_{12} \) reflect step \( b \), i.e., pattern \( sort \)’s behavior of generating formal expressions based on the clarified elements. Each transition reflects one of the rules in the \( \Psi \) item of the pattern \( sort \). For instance, transition \( p_{12} \rightarrow p_{13} \) generates formal expressions according to the rule \( tr_1 \).

Hierarchical relations between FSMs within an HFSM are used to describe the behavior of applying other patterns to formalize sub-functions. For example, the label of the transition \( p_{10} \rightarrow p_{11} \) involves pattern \( group \), which indicates that the interpretation of the transition needs the involvement of a low-level FSM. This low-level FSM is the top-level FSM of the HFSM corresponding to the pattern \( group \), which also uses lower-level FSMs to describe the details of certain components.

Based on the representation, the algorithm for utilizing the knowledge base can be given. Let \( q_0 \) be the initial state of the top-level FSM in the HFSM of the selected pattern, the utilization algorithm is given as follows where \( cs \) is an external variable denoting the current state of the HFSM and \( rs \) is an external variable denoting the sequence of high-level states.

\[
cs = q_0;
\]

\[
\text{if}(cs \text{ is an accept state})
\]

\[
\text{if}(rs \text{ is empty}) \quad \text{return no guidance};
\]

\[
\text{else} \quad cs = rs[rs.length - 1];
\]

\[
\text{for each transition } cs \rightarrow cs' \text{ originating from } cs
\]

\[
\text{if}(i = \text{processed input}&\text{each } g \in G \text{ is satisfied})
\]

\[
\text{cs} = cs';
\]

\[
\text{return } o;
\]

return error;

It starts from the initial state \( q_0 \) and displays guidance by traversing the states of the HFSM according to actual inputs from the developer. To deal with hierarchical relations in the HFSM, the following method is adopted. Given a upper-level FSM \( A \), for each component \( c \) in \( A \) interpreted by the set of lower-level FSMs \( LF \), it is utilized by traversing the FSMs in \( LF \).

In the developed prototype tool, the knowledge base is stored in a XML file with a set of pre-defined XML tags and the utilization algorithm is implemented to determine the tool’s behavior. The proposed approach is language-independent but its supporting tool can only work on a specific formal notation. Due to our expertise, we chose SOFL as an example formal notation and the prototype tool supports requirements formalization to construct SOFL formal specifications.

Figure 7 shows the main interface of the tool where the formal specification of the previously introduced bank data analysis system is under construction.

On the top left of the interface, a tree structure is given to demonstrate the architecture of the formal specification under construction. Users can edit the tree according to his design on the hierarchy of the involved modules. The example formal specification contains only one module \( Data\_Analysis \) which is denoted as the root node of the tree. On the right half of the interface, four editable text areas are used to display the content of the selected module where \( Type \) area denotes the defined data types, \( Var \) area denotes the defined variables, \( Processes \) area denotes the set of processes in the module and \( Inv \) area denotes the invariants in the module. When specifying the selected module, necessary types and variables are required to be first written. They will be automatically transformed into pre-defined data structures for producing formalization guidance. As can be seen from the figure, \( Data\_Analysis \) is the selected module where the types and variables are already declared. With well-defined types and variables, the specifying of the included processes and invariants can be started. Since the specifying of invariants under tool’s guidance is similar to that of the processes, we only present how the tool guides the writing of processes.

For each process, its inputs and outputs should be first defined manually. Then the functionality of the process can be described by writing pre- and post-conditions.
aided formalization is provided for users who find it difficult to directly write the pre-/post-condition in SOFL notation. It includes two tasks corresponding to the two steps for formalizing requirements based on the pattern system. The first task is to select an intended function from the “Function Selection” area on the bottom left of the interface. As the implementation of the pattern selection step, a tree structure is created in the “Function Selection” area to represent the hierarchy of the pattern categorization. The user can search for the intended function (pattern) from the root node to a leaf node and check the explanation of the function. In Fig. 7, the tool is supporting the writing of the previously introduced process `tran_Analysis` where its inputs and outputs are already defined by the developer. Function (pattern) `sort` is selected to aid the writing of the post-condition.

Once the selection decision is made by pressing “select”, a new interface will be displayed and the second task is to interact with the given guidelines through this interface. Figure 8 shows the interface for `sort` function consisting of two parts. The left part denotes the hierarchy of the guidelines in a guidance tree where high-level guidelines use low-level guidelines to deal with parts of their formalization problems, while the right part displays the detailed guidelines of the selected node. At the beginning, the tree structure is initialized with a root node denoting the top-level guidelines for formalizing sort functions and the first top-level guideline is shown. Since the previous case study has already explained how the guidelines are produced by applying the pattern `sort` to formalize process `tran_Analysis`'s behavior, it is skipped here and we focus on presenting the functionality of the tool.

As the response to the displayed guidance, formal expression “`account_store(in f).transactions`” denoting the set of transactions to be sorted is input to the given text box. After being confirmed by the developer, the response is submitted to the tool by pressing the “submit” button. The original interface is then refreshed as shown in Fig. 9 where the second top-level guideline is displayed below the first one. According to the interaction process presented in the case study, “`tranList`” is submitted as the response and the interface is refreshed again with a newly produced guideline that requires for the selection of the sorting rule’s category. Once the selection decision is made and submitted, the guidance for clarifying the content of the sorting rule is provided as shown in Fig. 10 (The included two snapshots show the details of the two nodes in the guidance tree respectively). It derives the sorting rule from two aspects: the rule for grouping transactions and the rule for sorting the grouped transactions. Since the guidance for the first aspect involves the application of the pattern `group`, node “aspect 1” is created as a child node of the root node “sort” to provide a clearer view on such an hierarchical relation. The detailed guidelines for specifying “aspect 1” is given in a new tab identified as “aspect 1” which can be activated by pressing the button “aspect 1” created in the original top-level guidance. In the figure, the top snapshot shows the top-level guidance for clarifying the sorting rule while the bottom snapshot shows the low-level guidance for specifying aspect 1.

Based on the understanding of process `tran_Analysis`’s behavior, the displayed graphical components can be filled out and submitted to the tool. After the tool receives the sub-
mitted information, the interaction process is finished and no more human effort is needed in the next formalization stage. With the function details derived from the interaction process, the tool automatically generates a formal expression as the post-condition of process `tran_Analysis` which is displayed in a new tab “result” as shown in Fig. 11. One can either directly copy the expression to the main interface or modify the expression in the tab before utilizing it.

4.2 Experiments

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

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 2 shows the result of the experiment. It summarizes 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. Column “Simplicity” denotes the rate \((ss/s) + (sg/g)/2\) 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.

![Fig. 11 Tool snapshot.](image)

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 paper. 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.

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, 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 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 3. As can be seen from the table, the average time for formalizing 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 significantly 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.

5. Related Work

There are mainly two kinds of requirements formalization methods. One kind is the methods for transforming from informal languages to formal notations. They allow developers to model the target system in graphical notations and formalize the graphical models 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 [10]. Cory Plock et al. show how to transform LSC (Live Sequence Charts) specifications with concurrency to timed automata [11]. Sunil Vadera et al. propose an interactive approach for producing formal specifications from English specifications [4]. Several tools have also been proposed for automatic transformation, such as U2B [12] and RoZ [13]. 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, but the introduction of patterns seems to offer a solution.

Some efforts have been made in the field of specification patterns. For informal specifications, several books [14], [15] 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. Meanwhile, researchers are still improving the usability of such patterns by specifying pattern solutions with pattern specification languages [16],

| Process behavior        | Average time for formalization | Average number of errors |
|-------------------------|--------------------------------|--------------------------|
|                         | group 1 | group 2 | group 1 | group 2 |
| customer authorization  | 4.5min  | 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                | 1.3min  | 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       |
For formal specifications, Stepney et al. describe a pattern language for using notation Z in computer system engineering [2]. The patterns proposed are classified into six types, including presentation patterns, idioms patterns, structure patterns, architecture patterns, domain patterns, development patterns. Each pattern provides a solution to a type of problem. Ding et al. propose an approach for specification construction through property-preserving refinement patterns [18]. Konrad et al. [19] 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.

Effective application of the above patterns requires full understanding, and the ability to select an appropriate one and solve specific problem using the suggested solution, because the representations make it impossible to utilize these pattern knowledge without human involvement. Consequently, although a large number of patterns have been developed, most of them are wasted as users may not fully understand how to leverage them in practice [20]. By contrast, our patterns serve as knowledge for machines to generate guidance and the formal definitions enable their automatic application.

Similar to us, authors in [21] also give formal definitions for their patterns 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.

6. Conclusion and Future Work

To assist practitioners in formalizing requirements, this paper proposes a pattern-based approach to guide the clarification of requirements and representation of the clarified requirements in formal expressions. A set of patterns are pre-defined in this approach and categorized in a hierarchy according to the functions they are used to formalize. A prototype tool that implements the approach is also described. It adopts HFSM to represent the pattern knowledge and implements the proposed algorithm for utilizing the pattern knowledge. Through two experiments, we have shown that the proposed approach is able to support computer-aided formalization of requirements. 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.

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.

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.

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.

We are also interested in developing techniques for au-
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.

Acknowledgement

This work has been conducted as a part of “Research Initiative on Advanced Software Engineering in 2012” supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA).

References

[1] S. Liu, Formal Engineering for Industrial Software Development, Springer, 2004.
[2] S. Stepney, F. Polack, and I. Toyn, “An outline pattern language for Z: Five illustrations and two tables,” ZB2003: Third International Conference of B and Z Users, Turku, Finland, ed. D. Bert, J.P. Bowen, S. King, and M. Walden, LNCS, vol.2651, pp.2–19, Springer, 2003.
[3] K. Miriyala and M. Harandi, “Automatic derivation of formal software specifications from informal descriptions,” IEEE Trans. Softw. Eng., vol.17, no.10, pp.1126–1142, Oct. 1991.
[4] S. Vadera and F. Meriane, “From English to formal specifications,” The Computer Journal, vol.37, no.9, pp.753–763, 1994.
[5] M. Elbendak, P. Vickers, and B.N. Rossiter, “Parsed use case descriptions as a basis for object-oriented class model generation,” J. Systems and Software, vol.84, no.7, pp.1209–1223, 2011.
[6] G. Cabral and A. Sampaio, “Automated formal specification generation and refinement from requirement documents,” J. Brazilian Computer Society, vol.14, no.1, pp.87–106, 2008.
[7] 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.
[8] L. Jin and H. Zhu, “Automatic generation of formal specification from requirements definition,” Proc. 1st International Conference on Formal Engineering Methods, ICFEM ’97, Washington, DC, USA, pp.243–251, 1997.
[9] D. Ilic, “Deriving formal specifications from informal requirements,” 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, pp.145–152, July 2007.
[10] W. McUmber and B. Cheng, “A general framework for formalizing uml with formal languages,” Proc. 23rd International Conference on Software Engineering, 2001, ICSE 2001, pp.433–442, May 2001.
[11] C. Plock, B. Goldberg, and L. Zuck, “From requirements to specifications,” 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS ’05, pp.183–190, April 2005.
[12] C. Snook and M. Butler, “U2B — A tool for translating UML-B models into B,” in UML-B Specification for Proven Embedded Systems Design, ed. J. Mermet, Springer, April 2004. Chapter: 6.
[13] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, “An overview of RoZ: A tool for integrating UML and Z specifications,” Proc. 12th International Conference on Advanced Information Systems Engineering, CAiSE ’00, pp.417–430, London, UK, 2000.
[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley Longman Publishing, Boston, MA, USA, 1995.
[15] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed., Prentice Hall, 2004.
[16] R. France, D.K. Kim, S. Ghosh, and E. Song, “A UML-based pattern specification technique,” IEEE Trans. Softw. Eng., vol.30, no.3, pp.193–206, March 2004.
[17] N. Soundarajan and J. Hallstrom, “Responsibilities and rewards: Specifying design patterns,” Proc. 26th International Conference on Software Engineering, ICSE 2004, pp.666–675, May 2004.
[18] J. Ding, L. Mo, and X. He, “An approach for specification construction using property-preserving refinement patterns,” SAC, pp.797–803, 2008.
[19] S. Konrad and B.H.C. Cheng, “Real-time specification patterns,” Proc. 27th International Conference on Software Engineering, ICSE ’05, New York, NY, USA, pp.372–381, 2005.
[20] D. Manolescu, W. Kozaczynski, A. Miller, and J. Hogg, “The growing divide in the patterns world,” IEEE Softw., vol.24, no.4, pp.61–67, 2007.
[21] J. Ackermann and K. Turowski, “A library of ocl specification patterns for behavioral specification of software components,” Proc. 18th International Conference on Advanced Information Systems Engineering, CAiSE ’06, pp.255–269, Berlin, Heidelberg, 2006.

Xi Wang is a Ph.D. student in the Department of Computer Science at Hosei University. She received her M.S. degree in the School of Computer Engineering and Science from Shanghai University, China, and Bachelor’s degree in the School of Information Engineering from Nanchang University, China. Her research interests include formal methods, formal engineering methods, formal specification and software engineering.

Shaoying Liu is a professor in the Department of Computer Science at Hosei University. He received his Ph.D. in formal methods at the University of Manchester, UK. His research interests include formal methods, formal engineering methods, software inspection, testing, and intelligent software engineering environments. He has published over 130 papers in refereed journals and international conferences and one book titled “Formal Engineering for Industrial Software Development” with Springer. He recently serves as a General Co-Chair for ICFEM 2012 and a PC member for many other conferences. He is a Fellow of British Computer Society, Senior Member of IEEE Computer Society, and member of Japan Society for Software Science and Technology.