A Pattern-Based Approach to Formal Specification Construction*

Xi Wang[1], Shaoying Liu[1], and Huaikou Miao[2]

[1] Department of Computer Science, Hosei University, Japan
[2] School of Computer Engineering and Science, Shanghai University, China

Abstract. Difficulty in the construction of formal specifications is one of the great gulfs that separate formal methods from industry. Though more and more practitioners become interested in formal methods as a potential technique for software quality assurance, they have also found it hard to express ideas properly in formal notations. This paper proposes a pattern-based approach to tackling this problem where a set of patterns are defined in advance. Each pattern provides an expression in informal notation to describe a type of functions and the method to transform the informal expression into a formal expression, which enables the development of a supporting tool to automatically guide one to gradually formalize the specification. We take the SOFL notation as an example to discuss the underlying principle of the approach and use an example to illustrate how it works in practice.

1 Introduction

Formal methods have made significant contributions to software engineering by establishing relatively mature approaches to formal specification, refinement, and verification, and their theoretical influence on conventional software engineering has been well known. The notion of pre- and post-conditions have been introduced into some programming languages to support the "design by contract" principle [1][2]; many companies have tried or actually used some formal methods in real software projects; and many practitioners have become more interested in formal methods nowadays than before (e.g., in Japan).

However, we also need to clearly understand the situation that despite tremendous efforts over the last thirty years in transferring formal methods techniques to industry, the real applications of formal methods in industry without academics' involvement are still rare, and many companies used them once and never return since, which are often not reported in the literature. For example, in Japan several companies have tried Z and VDM, but few of them have shown

* This work is supported in part by Okawa Foundation, Hosei University, Science and Technology Commission of Shanghai Municipality under Grant No. 10510704900 and Shanghai Leading Academic Discipline Project(Project Number: J50103). It is also supported by the NSFC Grant (No. 60910004) and 973 Program of China Grant (No. 2010CB328102).

a positive sign toward the future use of the same method again. One of the major reasons is that the practitioners find it hard to express their ideas properly in formal notations, not need to mention formal verification. This may sound difficult to accept for a formal method researcher or a person with strong mathematical background, but it is the reality in the software industry where vast majority of developers and designers may not even receive systematic training in computer science.

In this paper, we put forward a pattern-based approach to dealing with this challenge. It adopts the three-step approach in SOFL (Structured Object-oriented Formal Language) [3], which builds specification through informal, semi-formal and formal stages. Based on a set of inter-related patterns, guidance will be generated for specifying functions in terms of pre- and post-conditions step by step from semi-formal to formal stage, which enables developers to work only on the semantic level without struggling with complicated formal notations. These patterns are pre-defined, each providing an informal expression to describe certain function and the formalization method of such expression. Consider the statement "Alice belongs to student list", a corresponding pattern for "belong to" relation may suggest a formal expression, such as *Alice inset student_list* or *Alice in elems*(*student_list*), depending on the type of *student_list* being defined as a set type or sequence type (in the context of VDM and SOFL). This is only a simple example; the real application is of course more complex and would ask for further information from the developer during the construction process. Although researchers proposed many patterns to help handle commonly occurred problems on specification constructions, such as the well-known design patterns [4], developers have to understand the provided solutions before they can apply them to specific problems. By contrast, our patterns are hidden from developers; it is the understandable guidance that interacts with developers, which is produced based on patterns in our approach. We have also discussed the structure of individual patterns with an example, and demonstrated how the structure serves to establish well-defined informal expressions and formalize them by a real example. Such process is not expected to be fully automatic due to the need for human decisions, but it is expected to help the developer clarify ambiguities in the informal expression and select appropriate formal expressions.

Our proposed approach can be applied to any model-based formal specification notations. In this paper, we use SOFL language as an example for discussion. The reader who wishes to understand its details can refer to the book [3].

The remainder of this article is organized as follows. Section 2 gives a brief overview on related work. Section 3 describes the pattern-based approach. Section 4 presents an example to illustrate the approach. Finally, in Section 5, we conclude the paper and point out future research.

2 Related Work

Many reports about the use of patterns in specifications are available in the literature. For informal specifications, several books [4][5] 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 their usabilities by specifying pattern solutions with pattern specification languages [6][7]. For formal specifications, Stepney *et al.* describe a pattern language for using notation Z in computer system engineering [8]. The patterns proposed are classified into six types, including presentation patterns, idiom patterns, structure patters, architecture patterns, domain patterns, development patterns. Each pattern provides a solution to a type of problem. Ding *et al.* proposed an approach for specification construction through property-preserving refinement patterns [9]. Konrad *et al.* [10] created 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 offered a structured English grammar to facilitate the understanding of the meaning of a specification. This work is complementary to the notable Dwyer et al.'s patterns [11].

In spite of enthusiasm in academics, specification patterns are not yet widely utilized in industry mainly because of the difficulties in applying them. As can be seen from the related work, the patterns they established force their users to make a full understanding of them before selecting and applying appropriate ones. Statistical data shows that large amount of patterns have been developed, whereas users may not fully understand how to leverage them in practice [12]. Our approach treats patterns as knowledge that can be automatically analyzed by machines to generate comprehensible guidance for users. Thus, developers need neither to be educated on the patterns nor to be trapped in tedious and sophisticated formal notations; they can only focus on function design and make critical decisions on the semantic level.

3 Mechanism for Formal Specification Construction

3.1 Pattern Structure

Pattern is introduced to convey feasible solutions to re-occurred problem within a particular context, which is intended to facilitate people who face the same problem. Its structure varies depending on the specific situation it applies to and our pattern is defined as:

name	The name of the pattern
explanation	Situations or functions that the pattern can describe
constituents	A set of elements necessary for applying the pattern to write informal or formal statements
syntax	Grammar for writing informal expressions using the pattern
solution	Method for transforming the informal expression written using the pattern into formal expressions

To illustrate the structure, we explain each item through an example pattern shown in Figure 1 where $dataType(x)$ denotes the data type of variable x and $constraint(T)$ denotes certain property of variables of type T.

```
name: alter
explanation: For describing the change of variables or parts of variables
constituents:              semi-formal elements: obj
                           formal elements: decompose: Boolean, specifier, onlyOne: Boolean, new
  rules for guidance:
    1. if(dataType(obj) = basic type)  then  decompose = false
    2. if(deompose = false)  then  specifier = onlyOne = Null ∧ new = customValue ∧ dataType(new) = dataType(obj)
    3. dataType(obj)  →  specifier  /* determining the definition of specifier by the data type of the given obj */
         set of T (T is a basic type)     ① →   T | constraint(T) | specifier ∪ specifier
         set of T (T is a composite type with n fields f1, ⋯, fn)
                                          ② →   T | constraint(T) | constraint(fi) | specifier ∪ specifier   (1 ≤ i ≤ n)
                               T → T'     ③ →   (T → T' | constraint(dom) | constraint(rng) | constraint(dom, rng) |
                                                  specifier1 ∪ specifier1) * (dom | rng | dom ∧ rng)
              composite type with n fields f1, ⋯, fn
                                          ④ →   fi | specifier ∪ specifier    (1 ≤ i ≤ n)
                                           ……
    4. constraint(specifier)  →  onlyOne  /* determining the value of specifier by the given specifier */
           specifier(2) = dom ∧ rng     ① →    false
           specifier(1) = (dom = v) ∧ (specifier(2) = dom ∨ specifier(2) = rng)   ② →   true
                                           ……
    5. if(onlyOne = true)  then  (∃!x)(x ∈ obj ∧ specifier(x) ∧ dataType(new) = dataType(x))
        else  new is a mapping:  {x: obj | specifier(x)}  →  {originBased(p), customValue}
    ……     /* originBased(p): applying pattern p,  customValue: an input value */
 syntax: alter obj
 solution:  (dataType(obj), decompose, constraint(specifier), onlyOne, constraint(new))  →   formalization result
               initial ① →  obj = alter(˜obj) /* applied before specifying formal elements  if the pattern is not reused*/
               (any, false, any, any, dataType(new) = given)   ② →       new
               (any, false, any, any, new = originBased(p))   ③ →      p(obj)
               (set of T, true, any, true, any)   ④ →  let   x inset obj and specifier(x)
                                                       in   union(diff(obj, {x}), alter(x, false, Null, Null, new))
     (set of T, true, any, false, any)   ⑤→   "let X = {xi: obj | specifier(xi)}
                                               in  union(diff(obj, X)," forall[y: new] | "alter(xi, false, Null, Null, y)""")"
       composite type with n fields f_1, ⋯, f_n, true, any, false, any)
                        ⑥→    "modify(obj," forall[f_i: specifier] | "f_i → alter(obj.f_i, false, Null, Null, new(f_i))""")"
     (map, true, (specifier(1) = {(dom = v_1), ⋯, (dom = v_n)} ∧ specifier(2) = rng, false, any)
                        ⑦→    "override(obj, {" forall[x_i: specifier] | "v_i → alter(obj(v_i), false, Null, Null, new(x_i))""})"
                        ……
```

Fig. 1. Pattern "alter"

Item *name* is a unique identity based on which the pattern will be referenced. For the example pattern, the name "alter" is the identity.

Item *explanation* informally describes the potential effect of applying the pattern and suggests the situations where the pattern can be used. We can also find such item in pattern "alter", which tells that if one needs to depict modification on certain variable, pattern "alter" should be the choice.

Item *constituents* lists necessary elements to be clarified for writing expressions using the pattern and rules for guiding the clarification process. These elements can be divided into *semi-formal* and *formal* ones. The former are required to be designated when constructing well-defined informal expressions for semi-formal specifications while the latter only need to be specified during the formalization of the informal expressions. To avoid potential errors in specifications, each designated value v is required to be legal, which means v is a combination of constants and variables that can be found in the data definition part of the specification. After all the elements are legally specified, a *concrete constituents* is obtained where each element is bounded with a concrete value.

From Fig 1, we can see five elements listed in the *constituents* of pattern "alter". Element *obj* indicates the object intended to be modified; *decompose* is of boolean type meaning to replace the whole given *obj* by a new value if it is designated as true and to modify parts of the given *obj* if it is designated as false; $specifier$ denotes the description of the parts to be altered within *obj*; *onlyOne* is of boolean type meaning there exists only one part consistent with the description in $specifier$ if it is designated as true; *new* indicates the new values for replacing the corresponding parts to be altered.

The followed section "rules for guidance" includes two kinds of rules for guiding the process of specifying the listed five elements. One is represented as *if-then* statement that indicates certain conclusion when certain condition holds. For example, rule 1 states that if the given *obj* is of basic type, element *decompose* will be automatically set as false. This is obvious since a variable of basic type will not contain any smaller unit and therefore cannot be altered by replacing parts of it. The other kind of rules are represented as mappings, such as rule 3.

Item *syntax* establishes a standard format to write informal expressions using the pattern. For the example pattern, "*alter obj*" is valid to describe the modification on certain object when writing semi-formal specifications.

Item *solution* includes a method for producing a suggested formalization result according to the obtained *concrete constituents*. We define it as a mapping $constraint(constituents) \rightarrow string$ where $constraint(constituents)$ denotes properties of elements in *constituents*. Each property corresponds to a formalization result and a given *concrete constituents* that satisfies certain property will lead to the corresponding formalization result. For pattern *alter*, properties are represented through five sub-properties: the data type of *obj*, the value of *decompose*, properties of $specifier$, the value of *onlyOne* and properties of *new*. Let's take mapping 2 as an example where *any* denotes that whatever value is acceptable. It demonstrates the formalization result for all the *concrete constituents* satisfying the property that *decompose* is false and the value of the given *new* is of $customValue$ type.

As can be seen from pattern alter, some pattern names appear in the structure, such as the expression with "*alter*" in the *solution* item and "$originBased(p)$" in the *constituents* item, which means that the corresponding patterns will be reused when applying pattern "alter". Most reused patterns are attached with element information presented as: $p.name(arg_1, arg_2, ..., arg_m)(m <= n)$ where $p.name$ is the name of certain pattern p with n elements and $arg_1, arg_2, ..., arg_m$ are values of its first m elements listed in the *constituents* item.

3.2 Construction Method

Well-defined pattern structure establishes a foundation for our construction method which adopts the pattern notation for semi-formal specifications and applies the involved patterns to the formalization process. As shown in Fig 2, this method guides users through two stages: completion of semi-formal specifications and formalization of complete semi-formal specifications.

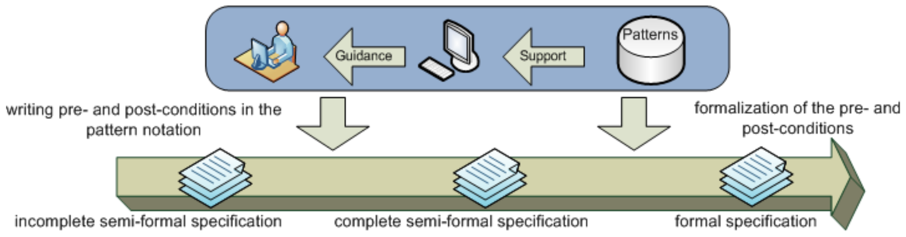
Guidance
Support
Patterns
writing pre- and post-conditions in the pattern notation
formalization of the pre- and post-conditions
incomplete semi-formal specification
complete semi-formal specification
formal specification

Fig. 2. Overview of the pattern-based approach

Since this paper focuses on building formal expressions, we assume that in incomplete semi-formal specifications, all the necessary types, variables, process names, process inputs and outputs are already defined. The main goal of the first stage is to help write pre- and post-conditions in pattern notation. Formal notations can also be used since they are able to logically combine single statements in pattern notation into a composite fragment to convey more complex meanings, meanwhile, the use of both languages allows expert users to choose a preferred one in writing specifications.

In the semi-formal stage, each pre- or post-condition is considered as a union of function units each described by certain pattern and different guidance will be given for different users towards building up each unit.

If the user is not familiar with the pattern notation, he will only be required to provide a general intention first by selecting patterns. This task is easy because: (1) pattern names are written in natural language and allows the selection to be easily done on semantic level, (2) the unique meaning of each pattern name differentiates itself from others and therefore avoids wrong selection, (3) the *explanation* items can help confirm the decision.

After a pattern p is selected, the user will be asked to specify its *semi-formal elements* in the order they are listed in the *constituent* item. The process of specifying each element e_i is as follows:

– If certain rules for determining the value of e_i have been activated, designate e_i according to the rules and begin to specify the element next to e_i.
– Otherwise, ask the user to specify e_i according to its definition given in p and the definition determined by certain activated rules if there is any. In case the designated value is illegal, the user will be informed and corrections can be done by defining the value as types or variables in the specification or replacing it with a legal one.
– All the rules in the "rules for guidance" item of p are examined with respect to the value of e_i and the activated ones are marked for later use.

Once the *semi-formal elements* of p have been specified, a corresponding informal expression is generated automatically on the basis of the *syntax* item of p and added to the corresponding pre- or post-condition.

Advanced users can choose to directly write expressions in pattern notation for pre- and post-conditions. Each atomic expression written with a pattern $p\prime$ will be checked based on the *syntax* item of $p\prime$ to extract information of *semi-formal elements*. For each element e designated as v:

1. If v is empty, i.e., value of e cannot be found in the expression, ask the user to specify e and insert the value to the expression according to syntax.
2. If v is illegal or does not conform to certain activated rules that determine the value or definition of e, ask the user to redefine v or replace it with a legal one conforming to the definition determined by all activated rules.
3. Examine all the rules in the "rules for guidance" item with respect to v and mark the activated ones for later use.

In this way, a valid expression can then be produced based on the original one and added to the corresponding pre- and post-condition.

One can choose from the above two alternative supporting strategies for each function unit and when each process of the specification is attached with a pre- and post-condition, the semi-formal specification is completed.

The user will be subsequently brought to the second stage where all the pre- and post-conditions are formalized sequentially. Since formal expressions are directly used for formal specifications, we only consider the formalization of expressions in pattern notation. For each atomic expression written with a pattern p, the user will be guided to specify formal elements of p by applying the same strategy for specifying *semi-formal elements*. Suppose that the obtained *concrete constituents* satisfies one of the conditions $constraint_i$ in the *solution* item of p, the expression will be formalized as $p.solution(constraint_i)$. The informal parts of the formalization result will be further formalized by repeating the above steps until no pattern notation is included.

In the same way, all the pre- and post-conditions can be formalized and the construction process will be finished resulting in the final formal specification.

4 Case Study: A Banking System

A relative small example of a banking system is used to illustrate our method. It provides four services for customers: *deposit*, *withdraw*, $information\ display$ and *currency exchange*. For the sake of space, we take the description process of the function *currency exchange* as an example which exchanges certain amount of source currency to corresponding amount of target currency on accounts.

Assume that the user has entered semi-formal stage and finished defining types and variables as indicated in Fig 3, as well as process names, inputs and outputs. Corresponding to the example function, process *Exchange* takes the type and amount of the source currency as inputs and produces a notice if the operation is successfully done. We take the most important function unit, updating account information, as an example to show how our approach works.

In case that the developer is not familiar with our pattern notation, he will be asked to select a pattern to express his general intention. Since the essential of the function unit is to modify an object of the system, the user can easily find the

```
type
Amount = real;
Date = nat0 * nat0 * nat0;
CurrencyType = {<JPY>, <USD>, <CNY>};
OperationType = {<deposit>,<withdraw>};
Balance = map CurrencyType to Amount;
Currency_pair = composed of
                    origin: CurrencyType
                    dest: CurrencyType
                   end;
Exchange_Rate = map Currency_pair
                   to real;
```

```
Transaction = composed of
                  date: Date
                  operationType: OperationType
                  currencyType: CurrencyType
                  amount: Amount
                 end;
Account = composed of
              number: string
              password: string
              balance: Balance
              transaction: seq of Transaction
             end;
```

```
var
ext #account_store:
      set of Account;
ext #today: Date;
ext #rate_store:
      Exchange_Rate;
```

Fig. 3. Definition of types and variables of the example specification

pattern *alter* as the best choice. As soon as the decision is made, the only *semi-formal element obj* is required to be clarified. The object to be altered is obvious and the user can also easily response with variable *account_store* which stands for the account datastore. Confirming that *account_store* is a legal input, an informal expression can be generated as "*alter account_store*" according to the *syntax* item of pattern *alter* and added to the semi-formal specification shown in Fig 4. Of course, advanced developers can directly write "*alter account_store*" in the semi-formal specification without guidance.

```
process Exchange(inf: string, currency: Currency_pair, amount: real) notice: string | warning: string
ext rd rate_store;
ext wr account_store;
post if (amount <= 0) then warning = "The required amount is invalid."
     else alter account_store and notice = "Successful operation"
end-process;
```

Fig. 4. Process *Exchange* in the semi-formal specification of the banking system

The complete semi-formal specification invokes the formalization process and formalizing process *Exchange* is actually formalizing "*alter account_store*". According to mapping 1 in the *solution* item of *alter*, the origin expression is transformed into $account\_store = alter(\tilde{~}account\_store)$ where $alter(\tilde{~}account\_store)$ needs further formalization. By the proposed method for specifying elements, Table 1 is obtained (column "reason" explains how the corresponding value generates). Notice that the obtained *concrete constituents* shown in Table 1 satisfies the property described in mapping 4 in the *solution* item; according to the formalization result it maps to, $alter(\tilde{~}account\_store)$ is formalized into:

$$\begin{array}{l} let\ x\ inset\ \tilde{~}obj\ and\ x.accountNo = inf \\ in\ \ union(diff(\tilde{~}obj, \{x\}), alter(x, false, Null, Null, originBased(alter))) \end{array}$$

There still exists an expression with "*alter*" which can be transformed into "$alter(x)$" according to mapping 3 in the *solution* item. Again, during the

Table 1. The designated formal elements for the expression $alter(\tilde{}account\_store)$

formal element	reason	value
$decompose$	altering specific accounts instead of the whole $account\_store$	true
$specifier$	"rules for guidance" rule 3 mapping 2	$accountNo = inf$
$onlyOne$	only one account is allowed to be active and receive currency exchange service at one time	true
new	"rules for guidance" rule 5	originBased($alter$)

Table 2. The designated elements for the expression $alter(x)$

element	reason	value
obj	already specified	x
$decompose$	altering balance and transaction of x instead of the whole x	true
$specifier$	"rules for guidance" rule 3 mapping 4	$\{balance, transaction\}$
$onlyOne$	there are two specific parts to be altered including field $balance$ and $transaction$	false
new	"rules for guidance" rule 5	$\{balance \rightarrow$originBased($alter$), $transaction \rightarrow$originBased($alter$)$\}$

formalization of $alter(x)$, Table 2 is derived. Mapping 6 in the *solution* item of $alter$ is then activated, which formalizes the expression $alter(x)$ as follows:

$$\begin{aligned}&modify(x, balance \rightarrow alter(x.balance, false, Null, Null, originBased(alter))\\&\qquad transaction \rightarrow alter(x.transaction, false, Null, Null, originBased(addTo)))\end{aligned}$$

Still, expressions involving $alter$ and $addTo$ (one of the patterns we designed) need to be formalized. By repeating the above procedures, process *Exchange* is formalized as shown in Fig 5.

```
process Exchange(inf: string, currency: Currency_pair, amount: real) notice: string | warning: string
ext rd rate_store;
ext wr account_store;
post if (amount <= 0) then warning = "The required amount is invalid."
     else account_store =
             let x inset ~account_store and x.number = inf
              in  union(diff(~account_store, {x}),
                        modify(x, balance → override(x.balance,
                                   currency.origin → x.balance(currency.origin) - amount)
                                   currency.dest → x.balance(currency.dest) + amount/rate_store(currency)),
                                  transaction → conc(x.transaction, [today, <withdraw>, currency.origin, amount],
                                                     [today, <deposit>, currency.dest, amount/rate_store(currency)])))
          and notice = "Successful operation"
end-process;
```

Fig. 5. Process *Exchange* in the formal specification of the example banking system

5 Conclusion

Trying to attack the obstacles to the application of formal specification techniques by practitioners in industry, this paper presents a method for guiding developers to gradually formalize semi-formal specifications based on specification patterns that are designed to be applied by machines to help formal specification construction. While we found this approach useful and effective, further improvements of the method are necessary.

Individual patterns need to be expanded and more patterns need to be created to handle more complex situations. Besides, the self-learning mechanism for updating the pattern-based knowledge base is also one of our future researches.

References

1. Meyer, B.: Eiffel: The Language. Prentice Hall Object-Oriented Series (1991)
2. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An Overview of JML Tools and Applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
3. Liu, S.: Formal Engineering for Industrial Software Development. Springer, Heidelberg (2004)
4. Gamma, E., Helm, R., Johnson, R.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
5. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edn. Prentice Hall (2004)
6. France, R.B., Ghosh, S.: A uml-based pattern specification technique. IEEE Transactions on Software Engineering 30, 193–206 (2004)
7. Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards : Specifying design patterns. In: 26th International Conference on Software Engineering, pp. 666–675 (2004)
8. Stepney, S., Polack, F., Toyn, I.: An Outline Pattern Language for Z: Five Illustrations and Two Tables. In: Bert, D., Bowen, J.P., King, S., Walden, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 2–19. Springer, Heidelberg (2003)
9. Ding, J., Mo, L., He, X.: An approach for specification construction using property-preserving refinement patterns. In: 23th Annual ACM Symposium on Applied Computing, pp. 797–803. ACM, New York (2008)
10. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: 27th International Conference on Software Engineering, pp. 372–381. ACM, New York (2005)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Pattern in property specifications for finite-state verification. In: 21th International Conference on Software Engineering, pp. 411–420. ACM, New York (1999)
12. Manolescu, D., Kozaczynski, W., Miller, A.: The growing divide in the patterns world. IEEE Software 24(4), 61–67 (2007)