Specification Refinement for Meta Code Patterns

Jian Liu, Farokh B. Bastani, I-Ling Yen
University of Texas at Dallas, Richardson, TX, 75083, USA

Abstract. Code pattern is proposed for component based software development (CBSD). A code pattern is used to capture the typical usages of a set of components as well as the interactions between them. Composition operations are defined to synthesize the glue code based on a repository of code patterns. Meta code patterns are patterns for components that share the same functionalities. In this paper, we briefly introduce the idea of code patterns. We discuss the pattern refinement process on meta code patterns and the map operation on code patterns. The refinement is to instantiate the meta code pattern to concrete pattern and then to code that is ready for use in the program. Signature refinement and functional refinement are discussed together with their application to meta code patterns. We use a case study to illustrate the basic idea and the process of how pattern refinement can lead to the final glue code from the meta code patterns.

1 Introduction

Software components are used for large-scale software development to reduce costs, risks, and time [1]. How to efficiently integrate the components is a major concern in this process. There has been research into methods of using components and automated glue code generation [2][3], but it has proved to be difficult. At the same time, effectively understanding and using components plays a significant role in component based software development. This includes understanding the circumstances that the components can be used in, the way the components can be composed together, and the constraints on the components.

The concept of design patterns and other software patterns, such as architecture patterns, has helped software developers resolve recurring problems encountered throughout all of the software development life-cycle and has enabled the integration of new systems from existing components [4][5][6]. The patterns hide the detailed interfaces of the components and document the design (interaction of the components) and the rationale behind them. While design patterns focus on the design of the system, not the usages of the components [4], we propose the use of code patterns to capture the typical usages of the components. Code patterns are code segments that capture the correct ways of using a set of components, especially the legal calling sequences for invoking the operations in the components [7]. Our approach uses code patterns to define flexible compositions.
and enable automated synthesis of the glue code by applying certain operations and transformations on code patterns [7].

Based on code patterns, meta code patterns are defined to catalog and generalize certain commonly recurring ways of using sets of components that share certain characteristics. Meta code patterns are patterns of code patterns. Their relationship is similar to that between classes and objects in object-oriented languages (OOL) in that a concrete code pattern is an instantiation of the meta code pattern. But a concrete code pattern is a refinement of the meta code pattern instead of a type or subtype as defined in the OOL. Since meta code pattern is an abstraction of multiple code patterns, if more meta code patterns can be found for the components, more concrete code pattern can be obtained using pattern refinement. This is useful since the more code patterns we have, the more will be the glue code that can be generated using the patterns.

In this paper, we focus on the definition of meta code pattern and its specification refinement and verification. During the pattern refinement process, signature and functional matching is used to find a suitable component for the code pattern. We formally define the signature and functional matching for the pattern refinement, and present a case study to illustrate the complete process.

The rest of this paper is organized as follows: In Section 2, we briefly discuss the definition of code patterns. Section 3 presents the meta code pattern and discusses its relationship with concrete code patterns. In Section 4, we introduce the refinement of meta code patterns. Section 5 presents a case study to show how code pattern refinement transforms meta code patterns into final code. Section 6 discusses the related work and Section 7 concludes the paper and identifies some future research directions.

2 Code Pattern Definition

The main purpose of code patterns is to provide a way for the developer to prevent errors when using a given set of components. It captures the typical calling sequences of the methods in the component. The definition and specification of code patterns are not the focus of this paper. In order to make the paper self-contained, we briefly discuss the pattern specification.

Each code pattern has a code template that is a formal specification of the structure and behavior of the solution. It has three major parts, namely, the interface, the pattern body and the constraints. The interface specifies what parameters the pattern has. Pattern body is a formal structure about the interaction between the components. The conditions needed to be satisfied in order to use the pattern are documented formally using OCL (Object Constraint Language) [8][9].

In the pattern interface, parameters are used to customize the pattern. There are two different types of parameters in code patterns. An actual parameter is a reference of a component. It denotes a specific object. A generic parameter represents the type of components or methods that have similar behaviors. It is an abstraction of these components and methods. The type definition we adapted
A type is either a type variable or a type operator applied to other types. There are two types of type operators, built-in operators and user-defined operators. An arity is associated with each type operator. For example, the "\( \rightarrow \)" constructor of the function types is binary, e.g., float \( \rightarrow \) int. Actual parameters can be viewed as the refinement of generic parameters.

Code pattern use two different parameter-passing mechanisms, namely, pass-by-name and pass-by-value parameters. Pass-by-value parameters are self-contained parameters that should be assigned specific values, while pass-by-name parameters are pieces of code that will be replaced by the passed content.

The following is the syntax definition of code template:

```
<CT>::=Code_template<interface><body>[<constraint>]End_code_template
@interface>::=Interface[<Name_para>][<Value_para>]End_interface
{Name_para}::=NAME:{[<para_type>]<para_name>[,<para_name>];}
{Value_para}::=VALUE:{<para_type><para_name>[,<para_type><para_name>];}
{para_type}::=[<language_type>][<System_type>]
{body}::=Body{<statement>[<nested_pattern>]<statement>}End_body
{nested_pattern}::=pattern <pattern_list>;{[| < nested_pattern >]
{pattern_list}::=pattern <pattern_list>[| < nested_pattern >]
{name_of_pattern}::=<name_of_pattern>[{(<pass_value>[,<pass_value>])}]
{pass_value}::=<para_name> = <para_value>
{constraint}::=Constraint{<OCL_statement>}End_constraint
```

Figure 1 shows the code template of the UPDATE_DATABASE pattern, which describes the typical way of updating a database table in Java: Create a database connection, execute the update statement, and close the statement and connection. The constraint section in the template formally specifies the conditions that need to be satisfied when applying this pattern.

Five pass-by-value parameters are defined in the code template: jdbcDriver, databaseURL, myLogin, myPassword, and smtString, which are the parameters of the methods of the components used in the pattern. The code template body specifies the calling sequence of the methods in those components as well as their interactions. The post-condition means the database is updated according to the corresponding SQL statement.

## 3 Meta Code Pattern

A meta code pattern is defined as a "pattern of code patterns". In component-based software development, it is common that certain groups of components share similar usage scenarios. And sometimes it may be difficult to identify code patterns for the components. Meta level code patterns provide a higher level abstraction of different concrete code patterns. A code pattern is a meta level pattern if one or more its parameters is a generic parameter. Typically, these generic parameters represent the types of the components or methods. By refining the generic parameters of meta code patterns to concrete components or methods, concrete code patterns for certain components can be generated.
Figure 2 is the code template of a meta code pattern that processes and deletes all the elements in an instance of an ADT (abstract data type), where $o \rightarrow asBag()$ represents the set of data in the ADT $o$ and $x \rightarrow asBag() - y \rightarrow asBag()$ means the set of data that are in $x$ but not in $y$. $o^A remove()$ is a variant of the standard OCL expressing, which represents the state of $o$ after the method call $o.remove()$. It is the opposite of the OCL $@pre$ construct and will be discussed in Section 4 in detail. The pre-condition of the pattern specifies what kinds of objects can be used in this meta pattern as well as the relationship between the methods of this object, which basically is the functional specification of the component that has these methods.

In some sense, meta code patterns are similar to design patterns in that they are both about the interaction of certain types of components and they both need to be instantiated or refined to be used. But the difference between them is also substantial: while design patterns focus on the design of system and the high level interactions between the components, the meta code pattern is mainly about how the components can be used and under what kind of context the interaction can be formed. Design pattern is about the design and meta code pattern is about using the components. The use of a design pattern starts from the design, and then finds the kind of the components that can be used in the design. The definition of a meta code pattern starts from certain kind of components, then observes what typical usages these components have.

Since the generic parameters, which are the types of the components, need to be instantiated for using a meta pattern, the user needs to define the signature

```java
Class.forName(_jdbcDriver);
Connection con = DriverManager.getConnection(_databaseURL, _myLogin, _myPassword);
Statement stmt = con.createStatement();
stmt.executeUpdate(_smtString);
stmt.close();
con.close();
```
and behavior of the desired components in detail. The behavioral specification of the code pattern as well as the parameters identifies what kinds of components can fit into the meta pattern and how to ensure that the desired post condition can be achieved. Meta pattern refinement is the process of finding correct refinements for the generic parameters, which will be discussed in Section 4.

4 Pattern Refinement

Refinement is defined as "formally and rigorously developing software from the specification" [11]. Meta pattern refinement is the process of obtaining a new pattern $p'$ by refining one or more generic parameters of a meta code pattern $p$ with more specific values. The refinement of meta code pattern $p$ generates a new pattern $p'$ with an increased descriptiveness. Meanwhile, the number of possible instantiations for a parameter is reduced with respect to those of the corresponding parameter in pattern $p$.

The types of the parameters (generic parameters) are instantiated during meta code pattern refinement. In a component based development environment, this means the types (generic parameters) are refined to the objects of certain concrete components. So pattern refinement here is mainly about to finding a function or component that can match the specification in the code pattern. The
refinement process checks if a given refinement matches the specification of the parameter. There will be two kinds of matching when the refinement is applied, namely, signature matching [12], and functional matching [13].

There are two different kinds of situations under which meta code pattern refinement is applied, namely, the method names are unknown but the type of the component is known and both the type of the component and the method names are unknown. In the first case, the type of the components is fixed and only the methods (type operator) are parameters. The second situation means that the parameters are types of components. If the type of the component is known, the signature and functional matching of the functions should be used in the refinement to find the corresponding methods. The component signature matching and component specification matching can be applied in the refinement when the type of the components is a parameter (generic parameter). Since component matching uses individual function matching, we discuss function refinement first. And since signature matching can always serve as a filter for the functional matching to exclude the candidates that do not have equivalent interfaces, we discuss signature matching first in each case.

**Individual function matching**

**Function signature matching** Signature matching is about matching function or component types. The composition of function signature matching in [12] is used in the refinement of the functions:

**Definition 1.** Function signature matching for code pattern:

\[
\text{Smatch}_p(\tau_p, \tau_{cp}(p)) = (\text{Smatch}_\text{gen} \circ \text{Smatch}_\text{reorder} \circ \text{Smatch}_\text{uncurry})(\tau_p, \tau_{cp}(p))
\]

where \(\text{Smatch}_p\) is the generalized signature match that the argument in the procedure is more general than those in the code pattern, \(\text{Smatch}_\text{reorder}\) is the reorder match that reorders the arguments of the procedure to match the interface, \(\text{Smatch}_\text{uncurry}\) does the uncurry transformation to the curried functions. \(p\) is the available function, \(cp(p)\) is the function specified in the code pattern, which normally is a generic parameter, \(\tau_p\) is the multiset of types in \(p\), \(\tau_{cp}(p)\) is the multiset of types corresponding to \(cp(p)\) in the code pattern.

The composition of signature matching used in pattern refinement must ensure that the procedure has the equivalent interface as the one specified in the code pattern. "Equivalent" here means that under certain transformations, the interfaces are identical.

**Function specification (functional) matching** The generalized predicate specification matching [13] is used when the functional specification of a function is checked.

**Definition 2.** Function functional matching for code pattern

\[
F\text{match}_p(p, cp(p)) = p_{\text{pred}} \Rightarrow cp(p)_{\text{pred}}
\]
where \( p_{\text{pred}} \) and \( \text{cp}(p)_{\text{pred}} \) are the specification of the available function and the corresponding generic parameter in the code pattern, respectively.

The generalized predicate specification match checking is used here because in code pattern refinement, we can focus on just the aspect of the behavior of the component that we are interested in, which is what is specified in the pattern. This means that any function that shares the same set of behaviors can be matched. Since the specification of the parameters is inside the pre-condition of the code pattern, the \( \text{cp}(p)_{\text{pred}} \) can be obtained by applying a projection function \( \text{fpred} \) to the pre-condition of the pattern:

\[
\text{cp}(p)_{\text{pred}} = \text{fpred}(\text{cp}(p), P)
\]

is defined as all the predicates in the pre-condition \( P \) that contains the corresponding parameter \( \text{cp}(p) \).

If the specification of the available function is a set of predicates \( S \), then \( S \) is \( p_{\text{pred}} \). If the specification is of pre- and post-condition format \((P, R)\), then the predicate \( p_{\text{pred}} = P \Rightarrow R \). For functions that specified using pre- and post-conditions in OCL, it is difficult to transform them to the format of \( P \Rightarrow R \). Because the OCL post-condition expression may have some context-sensitive definitions, such as \( @\text{pre} \), which represents the state at the start of the execution of the method, result, the notation of the return value of the method, and self, the reference to the contextual instance. These context-sensitive expressions can not be correctly interpreted without the corresponding context. In order to translate the OCL expression to a set of predicate, \( p_{\text{pred}} \), we use four rules during the transformation:

- Rule 1. Transform pre-condition \( P \) and post-condition \( R \) of the method to predicate \( P \Rightarrow R \).
- Rule 2. The keyword self in \( p_{\text{pred}} \) transforms to the component name that has the method.
- Rule 3. The keyword result in OCL transforms to the method name that returns the result.
- Rule 4. Instead of using OCL \( @\text{pre} \) to represent the start state, use \( \text{method()} \) in \( p_{\text{pred}} \) to represent the state after the execution of the method.

In order to distinguish the pre- and post-condition of the components and the pre- and post-condition of the code patterns, the specification of the pre-condition in the code pattern is also written using Rule 3 and Rule 4 (e.g., \textit{self} in code pattern specification means the code pattern itself) (Figure 2).

**Component matching**

We first define the interface of the components. A component’s signature interface is a pair, \((I_T, I_F)\), where \( I_T \) is a multiset of user-defined types, and \( I_F \) is a multiset of function types. A component’s specification interface is defined as a pair too, \( C=(C_T, C_F) \). \( C_T \) is a set of user-defined types and \( C_F \) is a set of function names each together with its functional specification. In component functional matching, the match of \( C_T \) can be ignored once \( C_F \) is matched since in order for \( C_F \) to be matched, \( C_T \) must be matched first using variable substitution [13].
Component signature matching  A variant of the generalized relax match of component defined in [12] is used for the pattern refinement.

**Definition 3. Component signature matching for code pattern**

\[ \text{Smatch}(I_c, I_{cp(c)}) = \exists \text{ a mapping } U_F: I_{cp(c)}F \rightarrow I_cF \text{ such that } U_F \text{ is one to one and } \forall \tau_{cp(p)} \in I_{cp(c)}F, \text{ Smatch}_p(U_F(\tau_{cp(p)}), \tau_{cp(p)}). \]

where \( I \) is the interface of the available component, \( I_{cp(c)} \) is the interface of the component specified in code pattern, and \( \text{Smatch}_p \) is the matching used in the single procedure matching for code patterns.

Component functional matching  A variant of the module matching in [13] is defined for the component functional matching check. The functional match of a component \((C_c)\) and a component specification in a code pattern \((C_{cp(c)})\) is defined as:

**Definition 4. Component functional matching for code pattern**

\[ \text{Fmatch}(C_c, C_{cp(c)}) = \exists \text{ total functions } U_CF \text{ and } U_CT, U_CF: C_{cp(c)}F \rightarrow C_cF, U_CT: C_{cp(c)}T \rightarrow C_cT \text{ (with corresponding to a sequence of variable renamings, } TC), \text{ such that } \]

- \( U_CF \) are one to one;
- \( \forall \tau \in C_{cp(c)}T, \text{ match}_E(\tau, U_CT(\tau)); \text{ and } \)
- \( \forall \tau \in C_{cp(c)}F, \text{ Fmatch}_p(U_CF(cp(p)), U_CT(cp(p))). \)

where \( C_{cp(c)}T \) and \( C_cT \) are user defined types, \( C_{cp(c)}F \) and \( C_cF \) are functional specifications, \( \text{match}_E(\tau, U_CT(\tau)) \) is the exact signature matching, which means after applying \( TC \) to \( \tau \), the obtained \( \tau_c \) is exactly matched with \( \tau \) [12]. The third property states that after the renaming process, for each function in the component specified in code pattern \((cp(p))\), there is a unique function in \( c \) that is matched to the specification with respect to \( \text{Fmatch}_p \).

From the discussion of Definition 2, if the specifications of the components and the parameters are treated as a set of predicates, then property three can be rewritten as

\[ \bigwedge_{i=1}^K U_CF(cp(p)_i) \Rightarrow \bigwedge_{i=1}^K U_CT(cp(p)_i) \text{ (K is the number of methods), } \]

where \( \bigwedge_{i=1}^K U_CF(cp(p)_i) \) is the conjunction of the predicates of all the methods that are in the component \( C \) and are the result of mapping \( U_CF(cp(p)_i) \). \( \bigwedge_{i=1}^K U_CT(cp(p)_i) \) is the conjunction of the predicates of applying the map \( U_CT \) to the method specification that the parameter has in \( C_{cp(c)} \).

The meta pattern refinement first uses the component signature matching to obtain components that have equivalent interfaces with the generic parameter in the pattern. Then it uses functional matching between the pattern specification and the components found in the first step to find the correct parameter refinement (the type of the component).

Section 5 uses the meta code pattern in Figure 2 and a component Queue to show the meta code pattern refinement process and its application.
5 Case Study

The meta code pattern in Figure 2 is about processing all the element in an ADT. Once the given parameter values can satisfy the pre-conditions, after the refinement, we can ensure that all the elements in this ADT are processed. We will use this meta pattern and a component Queue, whose functional behavior is specified in Figure 3 using OCL, to illustrate how the pattern refinement process refines a meta pattern by doing signature and functional matching between the specification of the component and the specification of the code pattern.

5.1 Signature Matching

In this case the signature matching is simple since both the component Queue and the parameter o have few methods and no method in the meta pattern has an argument. Using the component signature match (Definition 1), the head()
method of Queue matches $o$.retrieve(). The $o$.empty() is matched by the method empty() of Queue. The method remove() in the pattern can be matched by any method from empty(), head(), and deq() of Queue. There are totally $1 \times 1 \times 3 = 3$ different valid mappings ($U_F$). Figure 4 shows one of them. And this generally implies that the component Queue signature matches parameter $o$.

<table>
<thead>
<tr>
<th>$I_{cp(c)}$</th>
<th>$I_{cf}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o$.empty()</td>
<td>Queue.empty()</td>
</tr>
<tr>
<td>$o$.retrieve()</td>
<td>Queue.head()</td>
</tr>
<tr>
<td>$o$.remove()</td>
<td>Queue.deq()</td>
</tr>
</tbody>
</table>

Fig. 4. A signature mapping

Once the signature match finds the possible solutions, the component functional matching (Definition 4) is used to check the functionality of each signature mapping to see if the methods and components functionally match the parameters.

true implies self Queue\rightarrow asSequence()\rightarrow isEmpty() \quad (A)

true implies if self asSequence() isNotEmpty() then empty() = true \quad (B)
else empty() = false endif

not self asSequence() isNotEmpty() implies head() = self asSequence() (C)

not self asSequence() isNotEmpty() implies self deq() asSequence() (D)

true implies self enq(item) asSequence() = self asSequence() (E)

true implies if self asSequence() includes(o) then (F)
search(o) = self asSequence() indexOf(o) else search(o) = -1 endif

Fig. 5. Transformation of the Queue specification

5.2 Functional Matching

The first step of functional matching is to translate the component OCL specification using the four rules given in Section 4.1. The specification transformation
Fig. 6. Specification of o

Queue.empty() implies Queue\rightarrow asBag()\rightarrow isEmpty()
not Queue.empty() implies Queue\rightarrow asBag()\rightarrow includes(Queue.head())
not Queue.empty() implies
Queue\rightarrow asSequence()=Queue^{\text{deq}()\rightarrow asSequence()\rightarrow append(Queu.head())}

Fig. 7. Specification of mapping

of the Queue is shown in Figure 5. And B\land C\land D forms \bigwedge_{i=1}^{K} UCF(cp(p)_i) in Definition 4. The functional specification of the parameter o is found by applying the projection function on the pre-conditions of the pattern, \( cp(p)_{\text{pred}} = f_{\text{pred}}(cp(p), P) \), which is shown in Figure 6. It is not a specification method by method, but rather a specification of the relationship of the methods.

According to Definition 4, if we apply the mapping in Figure 4 to the specification of o (Figure 6), we get \( \bigwedge_{i=1}^{K} UCT(cp(p)_i) \) (Figure 7).

The next step is to verify if component Queue and its methods satisfy the parameter (o) specification, which is the process of verifying \( \bigwedge_{i=1}^{6} UCF(cp(p)_i) \Rightarrow \bigwedge_{i=1}^{6} UCT(cp(p)_i) \). In Figure 7, there are three predicates that need to be proved from the specification of Queue. The verification of the third predicate is shown in Figure 8. The verification of the other two predicates can be done using the same method.

This shows that the signature mapping in Figure 5 is functionally valid, which means the component Queue is a valid refinement of the parameter o in the meta code pattern \textsc{iterate}_{\text{ADT}}. This refinement generates the code pattern body for processing the elements in a Queue in Figure 9.

6 Related Works

Pattern refinement is a common issue in pattern-based approaches. [14] uses patterns to refine the obstacles (or constraints) in software requirements. Two refinement rules are defined: AND and OR. The patterns and the refinement rules are based on experience and there is no formal verification involved in the refinement when using these patterns. [15] uses patterns to design rules in workflows. Patterns are used as a generalized description of a recurring rule that can be associated with a workflow schema. Their refinement approach is essentially similar to ours. They use refinement-based and instantiation-based specification to rewrite the generic instructions.

The verification of the refinement is basically the type checking process. Step-wise refinement can be used to develop a complex program from a simple one by
not self→asSequence()→isEmpty() implies from (D) (1)
\[\text{deq()=self→asSequence()}→\text{first()}\text{and} \]
\[\text{self→deq()→asSequence()}=\text{self→asSequence()} \rightarrow \text{subSequence}(2,\text{self→asSequence().count()})\]

not self→asSequence()→isEmpty() implies from (1) (2)
\[\text{self→deq()→asSequence()}=\text{self→asSequence()} \rightarrow \text{subSequence}(2,\text{self→asSequence().count()})\]

self→asSequence()=self→asSequence() \rightarrow \text{append(a)}→\text{subSequence}(1,\text{self→asSequence()}) \rightarrow \text{append(a).count()-1)}

not self→asSequence()→isEmpty() implies from (3) and the (4)
\[\text{self→deq()→deq()→asSequence()}=\text{self→asSequence()} \rightarrow \text{append(self→asSequence()→first())}\]

not self→asSequence()→isEmpty() implies from (4) and (C) (5)
\[\text{self→deq()→deq()→asSequence()}=\text{self→asSequence()} \rightarrow \text{append(self.head())}\]

not self.empty() implies from (5) and (B) (6)
\[\text{self→deq()→deq()→asSequence()}=\text{self→asSequence()} \rightarrow \text{append(self.head())}\]

not Queue.empty() implies from (6) and (7)
\[\text{Queue→asSequence()}=\text{Queue→deq()→asSequence()} \rightarrow \text{append(Queue.head())}\]

Fig. 8. Verification process

adding features incrementally [16]. It is essentially a divide-and-conquer software development methodology. In each refinement step, new features are added. Some refinement methods apply refinement rules in each refinement step [17][18][19]. Some use refinement calculi to transform the specification into a concrete implementation in a more formal way [9][20][17][21]. Their refinements are different from pattern refinement in that they primarily deal with how to solve the problem, not mainly the structure of the code.

Architecture refinement is described in [5]. The verification strategy they use is basically similar to [22] where temporal logic of actions is used to formalize and prove the correctness. In a so-called parameterized refinement method [23], an extension of the refinement calculi, namely, probabilistic weakest pre-condition calculi [6], is used to determine and enhance the dependability of the program. Nonfunctional requirements are also considered in the refinement process.
Queue o;
while( !o.empty() ) {
    Object e = o.top();
    process(e);
    o.pop();
}

Fig. 9. Result of mapping

7 Conclusion and Future Research

We have introduced code patterns as a novel approach for component-based programming. Code patterns can help to capture the typical usages of the components and effectively deal with the problems encountered for more complex component compositions. We also introduced meta code patterns as patterns of concrete code patterns. The refinement of meta code pattern and its verification are described with examples to show how to derive a valid concrete code pattern from a meta pattern.

Research is needed in several directions to make code patterns more efficient and the refinement process more user-friendly. Currently, we assume that the refinement will be done manually. We wish to develop automated or semi-automated mechanism, such as theorem proving, to support the automation of pattern refinement and adaptation.

Since meta code patterns are patterns of code patterns, they can help to generate more useful code and help us to organize code patterns. We are exploring methods for automatically extracting meta code patterns from given specifications of component interactions. Applying program splicing techniques to component composition specifications may yield meta code patterns.

References