Knowledge-Based Support for Object-Oriented Software Design and Synthesis: a category theoretic approach

Yujun Zheng 1,2, Jinyun Xue 2, Qimin Hu 1,2
1 Institute of Software, Chinese Academy of Sciences
Beijing 100080, China.
yujun.zheng@computer.org
2 Center of Software Sci. & Tech., Coll. & Univ. of Jiangxi province
Nanchang 330027, China.
fjxue, qiminhu@jxnuic.e.com

ABSTRACT: To reuse previous knowledge of object-oriented design and adapt them to solve new problems, the collaboration relationships and the responsibility distribution among software objects need to be thoroughly understood and precisely formulated. The paper proposes a knowledge-based approach that employs category theoretic models to formalize and mechanize object-oriented software design and synthesis by focusing concern on reasoning about the interdependency relationships at different levels of abstraction and granularity. The major benefit of our approach is twofold: First, it provides an explicit semantics for formal object-oriented specifications, and therefore enables a high-level of reusability and dynamic adaptability. Second, it utilizes the ability of categorical computations to support automated software composition and refinement. A prototype tool that demonstrates the feasibility and effectiveness of our approach is also presented.

Categories and Subject Descriptors
D. 2.4 [Software/Program Verification]; D. 2.8 [Software Metrics] D.2.9 [Software Management]

General Terms
Software reliability, Software systems, Software quality and reliability

Keywords: Object oriented design, Automated software, Software quality, Software metrics

Received 30 November 2006; Revised 15 February 2007; Accepted 12 March 2007

1. Introduction

One way to software reuse is to construct and use source code libraries, which heavily rely upon directory information and keyword/string search that is often cumbersome and problematic [18]. Subsequently, many knowledge-based systems (e.g., [4, 9, 24]) have been investigated for more sophisticated support mechanisms, which typically have a knowledge representation of reusable software components, and provide knowledge-based functions to query, browse and edit the components to meet user requirements. The advantages of using a knowledge-base approach include: semantic retrieval and proliferation, information aggregation and intelligent index for components, and use of classification and inheritance to support updates.

However, users of those software reuse systems generally face with both a terminological and a cognitive gap [14]. Focusing concern on providing a formal definition of basic object-oriented concepts by extending current ADT-based specification language such as Z [23], B [1], and Slang [7], some formal models (e.g., [8, 11, 17]) have been proposed to facilitate the representation, understanding and validation of object-oriented knowledge, but they still lack a mathematically precise semantics for reasoning about the collaboration relationships and responsibility distribution among software objects, which are recognized as key to effective design and reuse of object-oriented design (OOD) frameworks in current component-based software development.

As a “theory of functions”, category theory offers a highly formalized language for object-oriented specifications, and is especially suited for focusing concern on reasoning about relations between objects. Also, it is sufficiently abstract that it can be applied to a wide range of different specification languages [26]. In this paper we present a knowledge-based approach that employs a category theoretic framework to model object-oriented software design and synthesis. This approach

- formally defines and represents the application domain knowledge with theory-based specifications at different levels of granularity;
- explicit models the similarities and variations among software components at different levels of abstraction;
- utilizes category theoretic computations to generate new software components out of existing ones automatically.

In consequence, our approach enables a high level of reusability and dynamic adaptability and features mechanizable specification composition, refinement and code generation. We also implement a prototype of a knowledge-based tool which has been successfully built into MISCE [27] to support object-oriented software development.

The remainder of the paper is structured as follows: Section 2 briefly introduces the basic notions of category theory. Section 3 describes the theory-based framework for formally representing object-oriented design and synthesis. Section 4 depicts the categorical operations for modeling the interdependency relationships among object-oriented designs and constructions for achieving the high-level reusability through theoretic computations. Section 5 presents an overview of our prototype tool with case studies of design pattern and domain-specific framework reuse. Section 6 concludes with discussion and future work directions.

2. Category Theory

Category theory, with its increasing role in computer science, has proved useful in the semantic investigation of programming languages [5]. First we introduce some basic notions of category theory, sufficient to understand the paper. Definition 1 A category $C$ is

- a collection $Ob$ of objects
- a collection $Mor$ of morphisms (arrows)
- an operation $id$ (identity) assigning to each object $b$ a morphism $id_b$ such that $dom(id_b) = cod(id_b) = b$
- an operation $\circ$ (composition) assigning to each pair $f, g$ of arrows with $dom(f) = cod(g)$ and arrow $f \circ g$ such that $dom(f \circ g) = dom(g)$, $cod(f \circ g) = cod(f)$
whose vertices and arrows with dom(f) = cod(g) and arrow f o g such that dom(f o g) = dom(g), cod(f o g) = cod(f).

Identity and composition must satisfy: (1) for any arrows f, g such that cod(f) = dom(g), we have id(dom) = f and g( id(dom) ) = g; (2) for any arrows f, g, h such that dom(f) = cod(g) and cod(g) = dom(h), we have (f o g) o h = f o (g o h).

Definition 2 A diagram $D$ in a category $C$ is a directed graph whose vertices $i \in I$ are labeled by objects $di \in ObC$ and whose edges $e \in E$ labeled by morphisms $f e : MorC$. A diagram $D$ is a family of morphisms $\{ fi : MorD \}$ such that (1) each arrow $f i : di \to dj$ in $D$, we have $f j o f i = f$. (2) for any arrows $f, g$ in $C$, we have $h o fn = f o (g o h)$.

Definition 3 Let $C$ be a category, $a; b \in ObC$ and $f : a \to b$.

- $f$ is epic iff, $g o f = h o f \Rightarrow g = h$
- $f$ is monic iff, $f o g = f o h \Rightarrow g = h$
- $f$ is isomorphic iff, there exists $g : b \to a$ such that $f o g = ida$ and $g o f = idb$

Definition 4 Let $D$ be a diagram in $C$, a cocone to $D$ is

- a $C$-object $x$
- a family of morphisms $(fi : di \to x \in I)$ such that for each arrow $g : di \to dj$ in $D$, we have $f j o g = fi$, as shown in Figure 1(a).

Definition 5 Let $D$ be a diagram in $C$, a colimit to $D$ is a $C$-object $x$ such that for any other cocone with $C$-object $x0$, there is a unique arrow $k : x \to x0$ in $C$ such that for each $di$ in $D$, $f f i : di \to x$ and $f fi : di \to x0$, we have $k o fi = f fi$ (as shown in Figure 1(b)).

Definition 6 An $\omega$-diagram in a category $C$ is a diagram with the structure shown in Figure 1(c).

Definition 7 Let $C$ and $D$ be categories, a functor $F : C \to D$ is a pair of operations $Fob : ObC \to ObD, Fmor : MorC \to MorD$ such that, for each morphism $f : a \to b, g : c \to d$ in $C$,

- $F Fob(f) : F Fob(a) \to F Fob(b)$
- $F Fmor(f o g) = F Fmor(f) o F Fmor(g)$
- $F Fmor(id) = id F Fmor(a)$

Definition 8 Let $C$ and $D$ be categories and $F, G : C \to D$ be functors, a natural transformation $\eta : F \to G$ is a function that assigns each object $a$ in $C$ a morphism $\eta a : Fob(a) \to Gob(a)$ in $D$, such that for each morphism $f : a \to b$ in $C$ we have $b o \eta a = \eta b o f$.

Definition 9 Given a family of morphisms $f1, f2, \ldots, fn \in Mor[a; b]$, a coequalizer is an object $e$ and a morphism $\epsilon : Mor[b; e]$ such that (1) $o f1 = o f2 = \cdots = o fn$; (2) for each $h \in Mor[b; c]$, $h o f1 = h o f2 = \cdots = h o fn$, there exists $k$ $Mor[c; e]$ such that $k o i = h$, as shown in Figure 1(e).

3.1 Basic Constructions

3.2 Object-Based Categories

Now we show how to use notions of category theory to abstract OOD at the levels of object, class, meta-class, and meta-meta-class. Following constructions are an extended version we previously developed in [28].

Definition 10 A signature $\Sigma = < S, \circ >$, where $S$ denotes a set of sort symbols, and $\circ$ denotes a set of operators. In more detail, $\Sigma = < C, F, P >$, where $C$ is a set of sorted constant symbols, $F$ a set of sorted function symbols, and $P$ a set of sorted predicate symbols.

Definition 11 $S$ is a category with signatures as objects, and a signature morphism is a consistent mapping from one signature to another.

Definition 12 A specification $SP = < \Sigma, \Phi >$, where $\Sigma$ is a signature, and $\Phi$ is a (finite) set of axioms over $\Sigma$.

Definition 13 $SPEC$ is a category with specifications as objects, and a specification morphism between specification $\Sigma 1 \to \Sigma 2$, where all the axioms in $\Phi 1$ are translated to theorems in $\Phi 2$.

3.3 Object-Based Categories

Now we show how to use notions of category theory to abstract OOD at the levels of object, class, meta-class, and meta-meta-class. Following constructions are an extended version we previously developed in [28].

Definition 14 An object signature $\alpha = < \alpha 1 : \Gamma >$, where $\alpha 1 = < S, Q >$ is a (universe) signature, $\alpha$ is an $S * X S$-indexed family of attribute symbols, and $\Gamma$ is an $S * X S$-indexed family of action symbols.

Definition 15 $OBJ-SIG$ is a category with object signatures as objects, and an object signature morphism is a consistent mapping from one signature to another.
Definition 16 An object specification \( OSP = \langle \varphi, \Phi \rangle \), where \( \varphi \) is an object signature, and \( \Phi \) is a (finite) set of \( \varphi \)-axioms.

Definition 17 \( OBJ\-SPEC \) is a category with object specifications as objects, and a morphism between specification \( \langle \varphi_1, \Phi_1 \rangle \) and specification \( \langle \varphi_2, \Phi_2 \rangle \) is a mapping of signature \( \varphi_1 \) into signature \( \varphi_2 \) such that all the axioms in \( \Phi_1 \) are translated to theorems in \( \Phi_2 \).

In the object-oriented world, there seem to be two different notions of class: a sort of abstraction and an extensional collection of objects [10]. Here we construct the category of classes out of the category of objects in the second sense, and so are the category of meta-classes and the category of meta-meta-classes.

Definition 18 Let \( D_1, D_2, \ldots, D_n \) be \( \omega \)-diagrams in \( OBJ\-SPEC \) and \( COL_i \) be colimits for \( D_i (i = 1; 2; \ldots; n) \), then \( CLS\-SPEC \) is a category with \( COL_i \) as objects, and a class morphism between \( COL_1 \) and \( COL_2 \) is the colimit of all morphisms in \( OBJ\-SPEC \) that between an object in \( D_1 \) and an object in \( D_2 \).

Definition 19 Let \( D_1, D_2, \ldots, D_n \) be \( \omega \)-diagrams in \( CLS\-SPEC \) and \( COL_i \) be colimits for \( D_i (i = 1; 2; \ldots; n) \), then \( M\-CLS\-SPEC \) is a category with \( COL_i \) as objects, and a meta-class morphism between \( COL_1 \) and \( COL_2 \) is the colimit of morphisms in \( CLS\-SPEC \) that between a class in \( D_1 \) and a class in \( D_2 \).

Definition 20 Let \( D_1, D_2, \ldots, D_n \) be \( \omega \)-diagrams in \( M\-CLS\-SPEC \) and \( COL_i \) be colimits for \( D_i (i = 1; 2; \ldots; n) \), then \( MM\-CLS\-SPEC \) is a category with \( COL_i \) as objects, and a meta-meta-class morphism between \( COL_1 \) and \( COL_2 \) is the colimit of morphisms in \( M\-CLS\-SPEC \) that between a meta-class in \( D_1 \) and a meta-class in \( D_2 \).

Functors from \( CLS\-SPEC \) to \( OBJ\-SPEC \) can be treated syntactically as instantiations or refinements. Similarly, we can consider a meta-class as a parameterized specification from a collection of class specifications, and a meta-meta-class from meta-class specifications. Thus functors from \( M\-CLS\-SPEC \) to \( CLS\-SPEC \) are the refinements from metaclasses to classes, and functors from \( MM\-CLS\-SPEC \) to \( M\-CLS\-SPEC \) the refinements from meta-meta-classes to meta-classes. Figure 2(a) illustrates the bottom-up way of object-oriented categories construction.

Definition 21 \( E\-OBJ\-SPEC \) is a discrete category with (executable) programs as objects. Functors from \( OBJ\-SPEC \) to \( E\-OBJ\-SPEC \) just take each \( O \in ObOBJ\-SPEC \) to (one of) its implementation(s) \( P \in ObE\-OBJ\-SPEC \), as shown in Figure 2(b).

3.3 Framework-Based Categories
Taking an OOD framework as a community of objects/classes/meta-classes/meta-meta-classes, it is straightforward to construct new categories out of object-based categories by composing their objects and relations (morphisms).

Definition 22 \( MM\-FRM\-SPEC \) is a category with diagrams in \( MM\-CLS\-SPEC \) as objects, and a morphism between two meta-meta-frameworks, namely \( MM\-FRM_1 \) and \( MM\-FRM_2 \), is the colimit of morphisms in \( MM\-CLS\-SPEC \) that between a meta-meta-class of \( MM\-FRM_1 \) and a meta-meta-class of \( MM\-FRM_2 \).

Definition 23 \( M\-FRM\-SPEC \) is a category with diagrams in \( M\-CLS\-SPEC \) as objects, and a morphism between two meta-frameworks, namely \( M\-FRM_1 \) and \( M\-FRM_2 \), is the colimit of morphisms in \( M\-CLS\-SPEC \) that between a meta-class of

![Figure 2. Constructions of object-oriented categories](image)

![Figure 3. Constructing framework-based categories](image)

As shown in the above section, a knowledge base of categorial specifications consists of two main types of representations: Individual software objects (including specifications of meta-meta-classes, meta-classes, classes, objects, and executable objects) and composed software designs (including specifications of meta-meta-frameworks, meta-frameworks, frameworks, implemented frameworks, and executable frameworks). Correspondingly, software reuse can be achieved by constructing:

- Morphisms between individual software objects.
- Functors between composed software designs.
Under this framework, category theoretic computations can be applied to support automatic reuse of software (specification) designs and refinements. First, the colimit operation can be used for constructing refinements mechanically. That is, an existing refinement (morphism) \( f: A_1 \rightarrow A_2 \) can be applied to a new specification \( S_1 \) by constructing a morphism \( e: A_1 \rightarrow S_1 \) which classifies \( S_1 \) as having \( A_1 \)-structure \([22]\). In consequence, the new specification \( S_2 \) can be obtained by computing the colimit of \( e \) and \( f \) instead of performing the refinement \( g \), as shown in Figure 4(a).

Second, the basic principle to specify a system is to build the specification for each component separately, and then use the colimit operation to compose them together. Consider the specifications (namely \( A_1 \) and \( A_2 \)) of two interactive components, the morphisms between them form a new specification, namely \( R \), which expresses the interdependency relationships between the two components. By computing the colimit of \( f_1: R \rightarrow A_1 \) and \( f_2: R \rightarrow A_2 \), the composed specification (namely \( S \)) is automatically worked out\(^1\), as shown in Figure 4(b). This approach not only allows us to reason about interactions between software objects, but also helps to maintain the traceability of each shared area that may evolves at a different rate \([26]\).

In conclusion, at the object level, it needs two colimit operations to generate the implementation of the composed specification from scratch, one for composition and one for refinement. The number of colimits needed at the class level, meta-class level, and meta-meta-class level are three, four, and five respectively. Such operations can be easily extended to the composition of more than two components. However, if the refinements of individual specifications already exist (e.g., in a knowledge base), the number of colimit operations needed to generate the implementation of the composed specification are all two, no matter at which level. That is, suppose the refinements \( h_1: A_1 \rightarrow A_1 \) and \( h_2: A_2 \rightarrow A_2 \) exist in the knowledge base, instead of performing a refinement \( S \rightarrow S' \), \( S' \) can be obtained by computing the colimit of \( f'_1: R' \rightarrow A'_1 \) and \( f'_2: R' \rightarrow A'_2 \), which are calculated by the composition \( h_1 \circ f_1 \circ e_1 \) and \( h_2 \circ f_2 \circ e_2 \) respectively (where \( e \) and its inverse \( e_1 \) can also be worked out automatically), as illustrated in Figure 4(c). Such constructions can also be easily scaled up to the paths through more than two levels of refinements.

\(^1\) For two interdependent specifications, the shared specification \( R \) is empty (a null object), and the colimit is just the combination of the two specifications.
4.2 Framework Reuse

By capturing successful OOD experiences, we can extract and abstract design knowledge, formally specifies the design (typically at the level of framework or meta-framework), rigorously implement them into executable products, and save the “standard” refinement history in the knowledge base. Next time encountering the similar problem in another context, instead of implementing a new refinement path, we can mechanically generate the new implementation via category theoretic computations.

As shown in Figure 5(a), taking a meta-framework $M$ from $M$-$FRM$-$SPEC$, if a refinement path $f_3 \circ f_2 \circ f_1$ already exists, by constructing a functor $h_1: FRM_1 \rightarrow FRM_2$, we can work out the refinement path for $FRM_2$ without refining $g_3 \circ g_2 \circ g_1$, manually. In detail, $g_1$ is the composition of $h_1 \circ f_1$, while $g_2$ and $h_2$ can be obtained by computing the colimit of $h_1$ and $f_2$, so are $g_3$ and $h_3$.

Furthermore, if the refinement $f_1: M \rightarrow FRM$, and $g_3: M \rightarrow FRM$ are both epimorphisms, $FRM$ itself can be constructed exclusively from $h$ and $FRM$, because $h$ is an isomorphism, which is proved as follows:

For morphisms $id: FRM \rightarrow FRM$, $id_2: FRM_2 \rightarrow FRM_2$, $h_1: FRM_1 \rightarrow FRM_2$, $h': FRM_2 \rightarrow FRM_3$, and $h''$: $FRM_3 \rightarrow FRM_4$, we have: (1) $id_1 \circ o \circ f = h''$, (2) $id_2 \circ o \circ g = h_1 \circ o = h_1 \circ o \circ f = h_3$. Since $f$ and $g$ are epic and therefore right cancelable, we have $h_1 \circ o \circ h_1 = id_1$ and $h_1 \circ o \circ h'' = id_2$, and thus $h_1$ is an isomorphism, as shown in Figure 5(b).

5. A Prototype Tool

We build a prototype tool that implements the knowledge-based approach for object-oriented software design and synthesis, whose architecture is shown in Figure 6. The tool is also a plug-in of MISCE, a domain-specific software development environment, which has been successfully applied in several industrial logistic information systems.

5.1 Knowledge Base

The knowledge base saves reusable OOD specifications at both the meta-framework level and the framework level together with their “standard” refinements. Such knowledge can be divided into the following three parts:

• Elementary types. The knowledge base contains an elementary library, which models a variety of abstract type specifications at the levels of meta-class and class, and implements their concrete types with C++ and C#. Those abstract types are also grouped into basic types (including simple types, aggregated types, and generic types) and domain-specific types (including entities, relations between entities, and constraints on those entities and relations). Users can query/browse existing types from the library, directly use them in software design, or generate new types by constructions illustrated in Section 4.1.

• Domain-specific frameworks. Any domain-specific language is referred to as a meta-model [19], or a meta-framework in category $M$-$FRM$-$SPEC$. In consequence, each model that abstracts one or more parts of a real-world system in the specific domain is an instance of the language, i.e., a framework in category $FRM$-$SPEC$. Based on the architectural design and/or reverse engineering of a number of logistic information systems [25, 30], we elaborately select about thirty architectural models that frequently occur in the logistic domain, construct their framework specifications, and save their refinement paths in the knowledge base for further reuse.

• Design patterns. A design pattern specification is the composition of its individual class-template (i.e., meta-class) specifications while all the properties are preserved [29]. In this sense, A design pattern can be viewed as a metaframework in $M$-$FRM$-$SPEC$. We select architectural patterns from [3, 12] and domain-specific patterns from [15], model their meta-framework specifications, refine them into executable programs, and save specifications together with refinements and final implementations in the knowledge base. Although users can directly reuse existing frameworks or patterns in software design, in most conditions they generate new instances for their specific contexts by constructions illustrated in Section 4.2.

5.2 Knowledge base editor

In the knowledge base, all the specifications and their refinements are formally described in CASL (common algebraic specification language) [6]. The tool provides an integrated GUI (see Figure 7) which contains an editor for users to manipulate the knowledge base. Currently, it allows the users to create or modify the specifications of elementary types and domain-specific frameworks through unified modeling language (UML) diagrams, and automatically generates/regenerates the relevant refinements. We are now

\[ h_3 \]

Although the morphisms $g_3$ and $h_3$ in Figure 5(a) can be worked out automatically via category theoretic computations, in most cases it is sufficient for us to generate the products at the level of implemented framework, since the refinement $g_3$ can always performed by compilers of programming languages, and $h_3$ is just an isomorphism between binary codes.
extending the editor to support creating design pattern specifications at the metaframework level. For domain-specific frameworks, the users can specify the following six types of relationships (morphisms) between individual class specifications:

- \( SCon\)\( (A_1; A_2) \): specification \( A_1 \) contains specification \( A_2 \).
- \( SCons\)\( (A_1; m_1; A_2) \): method \( m_1 \) of specification \( A_1 \) consumes field \( f_1 \) of \( A_2 \).
- \( FCons\)\( (A_1; m_1; A_2; f_1) \): method \( m_1 \) of specification \( A_1 \) consumes field \( f_1 \) of \( A_2 \).
- \( MCall\)\( (A_1; m_1; A_2) \): method \( m_1 \) of specification \( A_1 \) calls method \( m_2 \) of \( A_2 \).
- \( EDepe\)\( (A_1; e_1; A_2; e_2) \): event \( e_1 \) (invokes one or more methods) of specification \( A_1 \) arises after the event \( e_2 \) of \( A_2 \).
- \( Sinh\)\( (A_1; A_2) \): specification \( A_1 \) inherits specification \( A_2 \).
- \( MOve\)\( (A_1; m_1; A_2; m_2) \): method \( m_1 \) of specification \( A_1 \) overrides method \( m_2 \) of \( A_2 \).

### 5.3 Constructor and Generator

When reusing existing domain-specific frameworks or design patterns in the knowledge base, the users browse/query the library components, specify morphisms and functors, and construct new types, frameworks and patterns, which are all manipulated through UML diagrams and saved as background extensible markup language (XML) files. In the prototype tool, the pattern constructor and the domain framework constructor play the following three roles:

- Providing the GUI for the users to construct new designs by drag/drop and property alteration.
- Mapping UML diagrams in the GUI to background XML files [16].
- Parsing XML elements and building corresponding CASL-based specifications, which are sent to the program generator to generate executable code via category theoretic computations.

#### 5.3.1 Case Study 1: Domain Framework Reuse

First we show a simple example of domain-specific framework reuse through categorial constructions and refinements. Figure 8 illustrates the kernel part of a domain-specific language (meta-framework), namely TheManufactory, which has been refined into a system model (framework), namely M-FAC1. As shown in Figure 9(a), M-FAC1 refines each meta-class and meta-relationship of TheManufactory into exact one concrete class and relationship respectively. A "standard" model instance (implemented framework), namely IM-FAC1, is generated by building class instances into generic lists and relationship instances into class properties. The kernel code fragment (in C#) of IM-FAC1 is as follows:

```csharp
Golbal.Factories = new List<Factory>;
Factory fact1 = new Factory("Lumbermill");
fact1.Workers = new List<Worker>;
fact1.Machines = new List<Machine>;
Worker wor1 = new Worker("John");
fact1.Workers.Add(wor1);
fact1.Machines.Add(m1);
Golbal.Factories.Add(fact1);
```

Now suppose we need to refine the meta-framework TheManufactory into a new system model, namely M-FAC2, which has three instances of meta-class Machine and two instances of meta-class Worker, as described in Figure 9(b). To work out the implemented framework of M-FAC2, we need to construct a functor which specifies M-FAC2 has at least the categorical structure of M-FAC1. Such a functor \( H \) does exist, and it does not need to take each morphism in M-FAC1 to an M-FAC2-morphism; instead it just consists of the following parts that take three morphisms to three equalizers respectively (intra-framework morphisms are labeled in Figure 9):

- \( f_1 \rightarrow \) coequalizer\( (g_1; g_2; g_3) \)
- \( f_2 \rightarrow \) coequalizer\( (g_1; g_3) \)
- \( f_3 \rightarrow \) coequalizer\( (g_1; g_3) \)

In fact, here \( H \) is a homomorphism but does not need to be an isomorphism. Hence, by computing the colimit of \( H : M-FAC \rightarrow M-FAC \) and the library refinement \( F : M-FAC \rightarrow IM-FAC \), the new implemented framework namely IM-FAC2 can be worked out as follows:

```csharp
Golbal.Factories = new List<Factory>;
Factory fact1 = new Factory("Lumbermill");
fact1.Workers = new List<Worker>;
fact1.Machines = new List<Machine>;
Worker wor1 = new Worker("Mike");
fact1.Workers.Add(wor1);
fact1.Machines.Add(m1);
Machine mac1 = new Machine("Lather");
wor1.Machines.Add(mac1);
Mac1 = new Machine("Planer");
wor1.Machines.Add(mac1);
Golbal.Factories.Add(fact1);
```
5.3.2 Case Study 2: Design Pattern Reuse

Now we consider an example of design pattern reuse. As illustrated in Figure 10, the design pattern Observer from [12] is modeled as a meta-framework namely $M$-FRM, and its refinement in the knowledge base is $FRM_1$, where two observers DigitalClock and AnalogClock query the subject Timer to synchronize their time with the Time's state. To construct a new framework $FRM_2$ that represents the underlying dataset in separate forms of interface, we just construct a functor $H_1$:

$$FRM_1 \rightarrow FRM_2$$

This contains the following four parts:

- $f_0 \rightarrow g_0$
- $f_1 \rightarrow g_1$
- $\text{equalizer}(f_2; f_3) \rightarrow \text{equalizer}(g_2; g_3; g_4)$
- $\text{equalizer}(f_4; f_5) \rightarrow \text{equalizer}(g_5; g_6; g_7)$

Thereby, the new implemented framework $I$-FRM$_2$ is generated without refining $G_2$ of $F_2$, manually: $G_2$ is the composition $H_1 \circ F_1$, while $G_1$ and $H_2$ are obtained by computing the colimit of $H_1$ and $F_2$.

6. Conclusion

Mylopoulos [20] once argued that one important role that artificial intelligence can play in software engineering is to act as a source of ideas about representing knowledge that can improve the state-of-the-art in software information management, rather than just building intelligent computer assistants. To reduce the complexities inherent in large-scale software systems and improve reusability of previous software designs, knowledge-base techniques need to employ a mathematically precise semantics for reasoning about the interdependency relationships and responsibility distribution among software components.

We propose here a knowledge-based software development approach that employs category theoretic models to formalize and mechanize object-oriented software design and synthesis. The major benefit of our approach is twofold: First, it provides an explicit semantics for formal object-oriented specifications, and therefore enables a high-level of reusability and dynamic adaptability. Second, it utilizes the ability of categorical computations to support automated software composition and refinement at different levels of abstraction and granularity. The prototype tool that
implements our approach is also briefly introduced, with case studies of design pattern and domain-specific framework reuse. Our ongoing efforts include extending the category theoretic computations to other software artifacts including documents, test cases and scripts, and using a UML profile based formalism [2] to serve as a communication basis between experts and knowledge engineers during knowledge acquisition.

7. Acknowledgements

The work was supported in part by grants from NNSF (No. 60573080) and NGFR 973 Program (No. 2003CA02800) of China. We are indebted to the anonymous reviewers of ISDA’06 for their valuable suggestions to improve the content and the structure of the paper.

References