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, qiminhug@jxnucie.com

Journal of Digital Information Management

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 objectoriented 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 *ObC* of objects

– a collection *MorC* of morphisms (arrows)

– an operation *id* (identity) assigning to each object *b* a morphism *idb* such that *dom*(*idb*) = *cod*(*idb*) = *b*

– an operation ± (composition) assigning to each pair *f*, *g* of arrows with *dom*(*f*) = *cod*(*g*) and arrow *f* ± *g* such that

$$dom(f \pm g) = dom(g), cod(f \pm g) = cod(f)$$

– an operation *o* (composition) assigning to each pair *f*, *g* of 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*) = *b* = *dom*(*g*), we have *idbof* = *f* and *goidb* = *g*; (2) for any arrows *f*, *g*, *h* such that *dom*(*f*) = *cod*(*g*) and *dom*(*g*) = *cod*(*h*), we have (*f og*)*oh* = *f o*(*goh*)

Definition 2 A diagram *D* in a category *C* is a directed graph whose vertices *i* ∈ *I* are labeled by objects *di* ∈ *ObC* and whose edges *e* ∈ *E* labeled by morphisms *fe* ∈ *MorC*.

Definition 3 Let *C* be a category, *a; b* ∈ *ObC* and *f* : *a* → *b*:
– *f* is epic *iff*, *g o f* = *h o f* ⇒ *g* = *h*
– *f* is monic *iff*, *f o g* = *f o h* ⇒ *g* = *h*
– *f* is isomorphic *iff*, there exists *g* : *b* → *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* → *x* *i* ∈ *I }* such that for each arrow *g*: *di* → *dj* in *D*, we have *fj o g* = *fi*, as shown in Figure 1(a).

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

Definition 6 An ω**-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* → *D* is a pair of operations *Fob*: *ObC* → *ObD*, *Fmor*: $Mor_C \rightarrow Mor_D$ such that, for each morphism *f*: $a \rightarrow b$, g: $c \rightarrow d$ in *C*,
– $F_{mor}(f)$: $F_{ob}(a) \rightarrow F_{ob}(b)$
– $F_{mor}(f \text{ o } g) = F_{mor}(f) \text{ o } F_{mor}(g)$
– $F_{mor}(id_a) = idF_{ob}(a)$

Definition 8 Let *C* and *D* be categories and *F*, *G*: *C* → *D* be functors, a natural transformation : *F* → *G* is a function that assigns each object a in *C* a morphism ηa : *Fob*(*a*) → *Gob*(*a*) in D, such that for each morphism *f*: *a* → *b* in C we have η *b o* $F_{mor}(f) = G_{mor}(f)$ *0* η *a*, as shown in Figure 1(d).

Definition 9 Given a family of morphisms *f*1, *f*2 .... *fn* ∈ *Mor*[*a; b*], a coequalizer of them is an object *e* and a morphism *i* ∈ *Mor*[*b; e*] such that (1)*i o f*1 = *i o f*2 = : : : = *i o fn*; (2) for each *h* ∈ *Mor*[*b; c*], *h o f*1 = *h o f*2 = : : : = *h o fn*, there exists *k* *Mor*[*e; c*] such that *k o i* = *h*, as shown in Figure 1(e).

3. Design and Synthesis Framework

As mentioned above, category theory studies "objects" and "morphisms" between them: objects are not collections of "elements", and morphisms do not need to be functions between sets; any immediate access to the internal structure of objects is prevented. This is quite similar to the concepts in object-oriented methodology. Moreover, the objects of a metacategory correspond exactly to its identity arrows, and hence it is technically possible to dispense altogether with the objects and deal only with arrows (that is, the subject could be described as learning how to "live without elements" [21]). In this section, we lay down the foundations of object-oriented categorial framework, on top of which OOD knowledge are modeled and implemented. More details on the proof theory and the connections with temporal logic can be found in [11].

3.1 Basic Constructions

Category theory has been proposed as a framework for synthesizing formal specifications based on works by Goguen [13]. Following are basic notions of category localizations, in which a specification is the finite presentation of a theory, and a signature provides the vocabulary of a specification.

Definition 10 A signature Σ=< *S*, Ω >, where *S* denotes a set of sort symbols, and - denotes a set of operators. In more detail, Ω =< *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 *SIG* 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* =< Σ ; Φ >, where Σ is a signature, and Φ is a (finite) set of axioms over Σ .

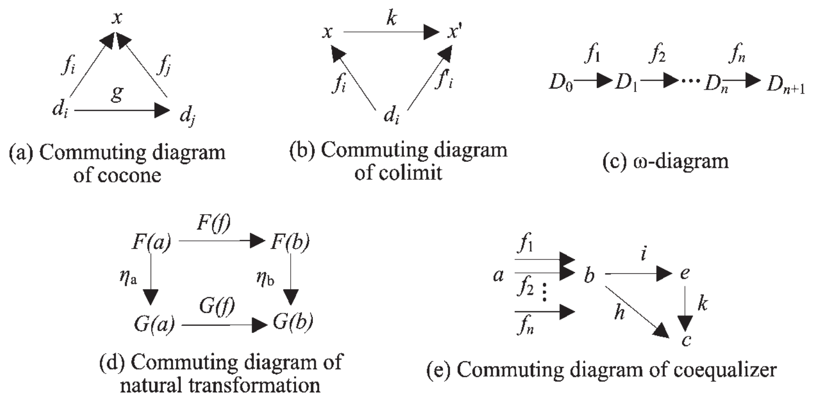
Definition 13 *SPEC* is a category with specifications as objects, and a specification morphism between specification <Σ_1; Φ_1> and specification <Σ_2; Φ_2> is a mapping of signature Σ_1 into signature Σ_2 such that all the axioms in Φ_1 are translated to theorems in Φ_2.

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 14 An object signature Θ =< Σ *; A;* Γ >, where Σ =< S, Ω > is a (universe) signature, *A* is an S^* X S-indexed family of attribute symbols, and Γ is an 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.



(a) Commuting diagram of cocone
(b) Commuting diagram of colimit
(c) ω-diagram
(d) Commuting diagram of natural transformation
(e) Commuting diagram of coequalizer

Figure 1. Basic diagrams in category theory

Definition 16 An object specification *OSP* =< Σ; Φ>, where Σ is an object signature, and Φ is a (finite) set of Σ -axioms.

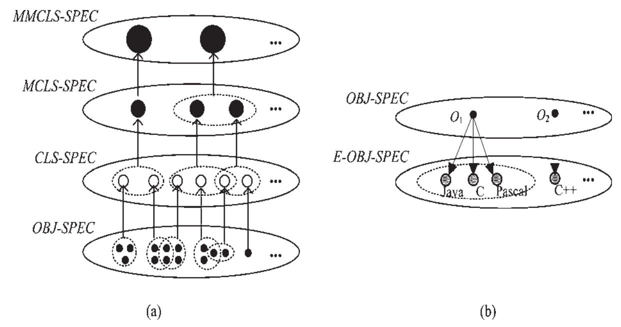


Figure 2. Constructions of object-oriented categories

Definition 17 *OBJ-SPEC* is a category with object specifications as objects, and a morphism between specification < Σ_1; Φ_1 > and specification < Σ_2; Φ_2> is a mapping of signature Σ_1 into signature Σ_2 such that all the axioms in Φ_1 are translated to theorems in Φ_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 . . .*Dn* be ω -diagrams in *OBJ-SPEC* and COL_i be colimits for D_i ($i = 1; 2 : : : n$), then *CLSSPEC* is a category with *COLi* 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 . . .D_n be ω -diagrams in *CLS-SPEC* and COL_i be colimits for D_i($i = 1; 2 : : : n$), then *M-CLS-SPEC* is a category with *COLi* 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 . . .D_n be ω -diagrams in *M-CLS-SPEC* and COL_i be colimits for D_i($i = 1; 2 : : : n$), then *MM-CLS-SPEC* is a category with *COLi* 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-metaclass 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$MM\text{-}FRM_1$ and$MM\text{-}FRM_2$, is the colimit of morphisms in *MM-CLS-SPEC* that between a meta-meta-class of $MM\text{-}FRM_1$ and a meta-meta-class of $MM\text{-}FRM_2$.

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

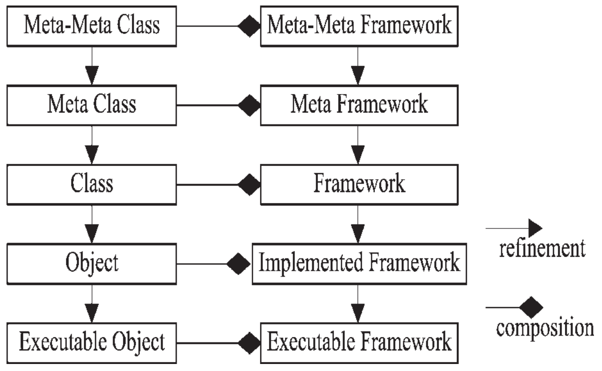


Figure 3. Constructing framework-based categories

$M\text{-}FRM_1$ and a meta-class of $M\text{-}FRM_2$.

Definition 24 ***FRM-SPEC*** is a category with diagrams in *CLS-SPEC* as objects, and a morphism between two frameworks, namely FRM_1 and FRM_2, is the colimit of morphisms in *CLS-SPEC* that between a class of FRM_1 and a class of FRM_2.

Definition 25 ***I-FRM-SPEC*** is a category with diagrams in *OBJ-SPEC* as objects, and a morphism between two implemented frameworks, namely $I\text{-}FRM_1$ and $I\text{-}FRM_2$, is the colimit of morphisms in *OBJ-SPEC* that between an object of $I\text{-}FRM_1$ and an object of $I\text{-}FRM_2$.

Definition 26 ***E-FRM-SPEC*** is a discrete category with (executable) programs as objects, and that functors from *I-FRM-SPEC* to *E-FRM-SPEC* just take each$O \in O_{bI\text{-}FRM\text{-}SPEC}$ to (one of) its implementation $P \in Ob_{E\text{-}FRM\text{-}SPEC.}$

As illustrated in Figure 3, the left refinement process of class-based specifications is brought to the right refinement process of framework-based specifications through compositions at different levels of granularity.

4. Operations and Constructions

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.

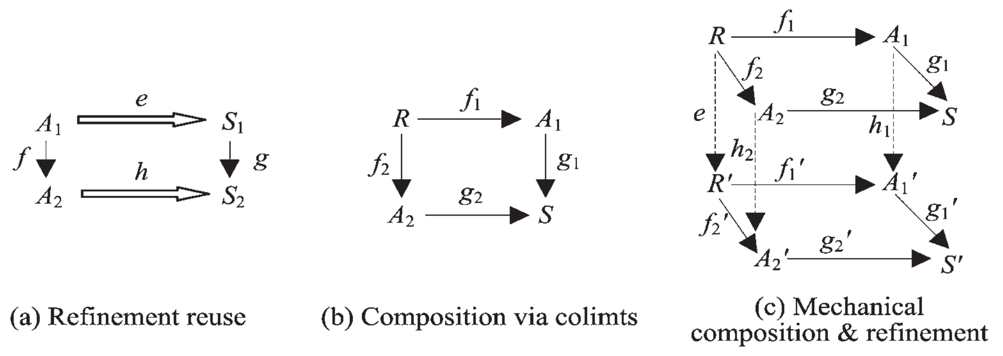


Figure 4. Modular refinement and composition

4.1 Composition and Refinement

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: $A1 \rightarrow A2$ can be applied to a new specification $S1$ by constructing a morphism e: $A1 \rightarrow S1$ which classifies $S1$ as having $A1$-structure [22]. In consequence, the new specification $S2$ 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 $A1$ and $A2$) 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 $f1$: $R \rightarrow A1$ and $f2$: $R \rightarrow A2$, 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 $h_1\ o\ f_1\ o\ e\text{-}_1$ and $h_2\ o\ f_2\ o\ e\text{-}_1$ respectively (where e and its inverse $e\text{-}_1$ can also be worked out automaticlly), as illustrated in Figure 4(c). Such constructions can also be easily scaled up to the paths through more than two levels of refinements.

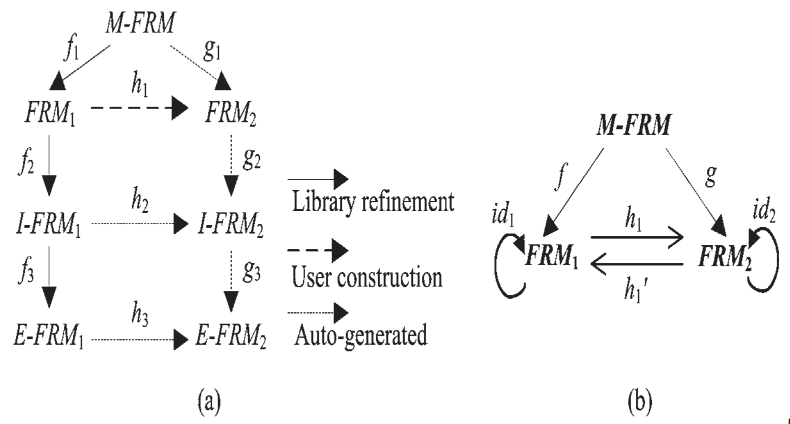


Figure 5. Implement a new framework by categorical computations

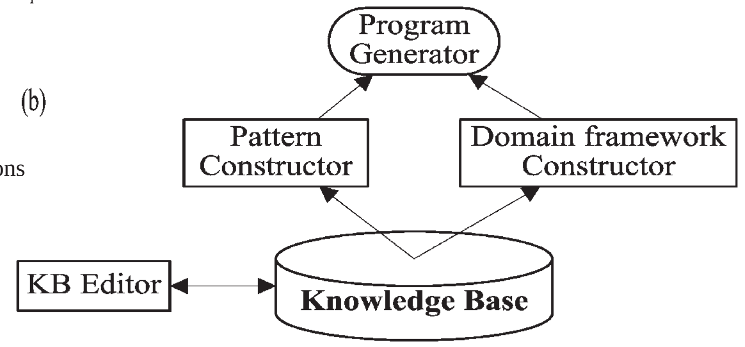


Figure 6. The architecture of the prototype tool

[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-FRM* from *M-FRM-SPEC*, if a refinement path $f3 \circ f2 \circ f1$ already exists, by constructing a functor $h1: FRM1 \to FRM2$, 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.[2]

Furthermore, if the refinement $f1: M\text{-}FRM \to FRM_1$ and $g_1: M\text{-}FRM \to FRM_2$ are both epimorphisms, FRM_2 itself can be constructed exclusively from h_1 and FRM_1 because h_1 is an isomorphism, which is proved as follows:

For morphisms $id_1: FRM_1 \to FRM_1$, $id2: FRM_2 \to FRM_2$, $h_1: FRM_1 \to FRM_2$ and $h'_1: FRM_2 \to FRM_1$, we have: (1) $id_1 \circ f = f = h'_1 \circ g = h'_1 \circ h_1 \circ f$; (2) $id_2 \circ g = g = h_1 \circ f = h_1 \circ h'_1 \circ g$. Since *f* and *g* are epic and therefore right cancelable, we have $h'_1 \circ h_1 = id_1$ and $h_1 \circ h'_1 = id_2$, 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).

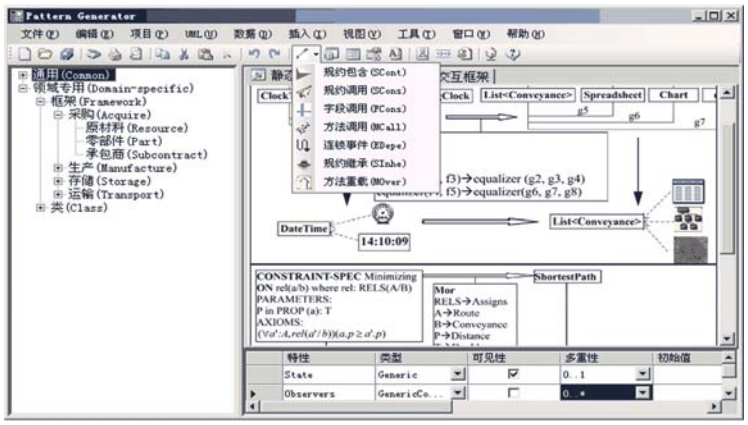
Users can query/browse existing types from the library, directly use them in software design, or generate new types by constructions illustrated in Section 4.1.

• 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



CONSTRAINT-SPEC Minimizing
ON rel(a/b) where rel: RELS(A/B)
PARAMETERS:
P in PROP (a): T
AXIOMS:
$(\forall a':A, rel(a'/b))(a.p \geq a'.p)$

Mor
RELS→Assigns
A→Route
B→Conveyance
P→Distance

特性	类型	可见性	多重性	初始值
State	Generic	✓	0..1	
Observers	GenericCo...	☐	0..*	

[2] 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:

• $SCont(A_1;A_2)$: specification $A1$ contains specification $A2$.
• $SCons(A_1;m1;A2)$: method $m1$ of specification $A1$ consumes $A2$.
• $FCons(A_1;m_1;A_2;\ f_2)$: method m_1 of specification A_1 consumes field f_2 of A_2.
• $MCall(A_1;m_1;A_2;m_2)$: method $m1$ of specification $A1$ 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.
• $SInhe(A_1;A_2)$: specification A_1 inherits specification A_2.
• $MOver(A_1;m_1;A_2;m_2)$: 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 send 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 illustrate the kernel part of a domain-specific language (meta-framework), namely *TheManufactory*, which has been refined into a system model (framework), namely*M-FAC*1. As shown in Figure 9(a),*M-FAC*1 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-FAC*1, is generated by building class instances into generic lists and relationship instances into class properties. The kernel code fragment (in C#) of *IM-FAC*1 is as follows:

```
Golbal.Factories = new List<Factory>;
Factory fact1 = new Factory(“Factory1”);
fact1.Workers = new List<Worker>;
fact1.Machines = new List<Machine>;
Worker wor1 = new Worker(“Worker1”);
fact1.Workers.Add(wor1);
wor1.Machines = new List<Machine>;
Machine mac1 = new Machine(“Machine1”);
wor1.Machines.Add(mac1);
fact1.Machines.Add(mac1);
Golbal.Factories.Add(fact1);
```

Now suppose we need to refine the meta-framework *TheManufactory* into a new system model, namely *M-FAC*2, 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-FAC*2, we need to construct a functor which specifies*M-FAC*2 has at least the categorial structure of*M-FAC*1. Such a functor *H* does exist, and it does not need to take each morphism in*M-FAC*1 to an *M-FAC*2-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):

• $f1 \rightarrow coequalizer(g_1;\ g_2;\ g_3)$
• $f2 \rightarrow coequalizer(g_4;\ g_5)$
• $f3 \rightarrow coequalizer(coequalizer(g_6;\ g_7);\ g_8)$

In fact, here *H* it is a *homemorphism* but does not need to be an *isomorphism*. Hence, by computing the colimit of $H : M\text{-}FAC_1 \rightarrow M\text{-}FAC_2$ and the library refinement $F : M\text{-}FAC_1 \rightarrow IM\text{-}FAC_1$, the new implemented framework namely $IM\text{-}FAC_2$ can be worked out as follows:

```
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);
wor1.Machines = new List<Machine>;
Machine mac1 = new Machine(“Lather”);
wor1.Machines.Add(mac1);
fact1.Machines.Add(mac1);
mac1 = new Machine(“Slicer”);
wor1.Machines.Add(mac1);
fact1.Machines.Add(mac1);
wor1 = new Worker(“John”);
fact1.Workers.Add(wor1);
wor1.Machines = new List<Machine>;
mac1 = new Machine(“Planer”);
wor1.Machines.Add(mac1);
fact1.Machines.Add(mac1);
Golbal.Factories.Add(fact1);
```

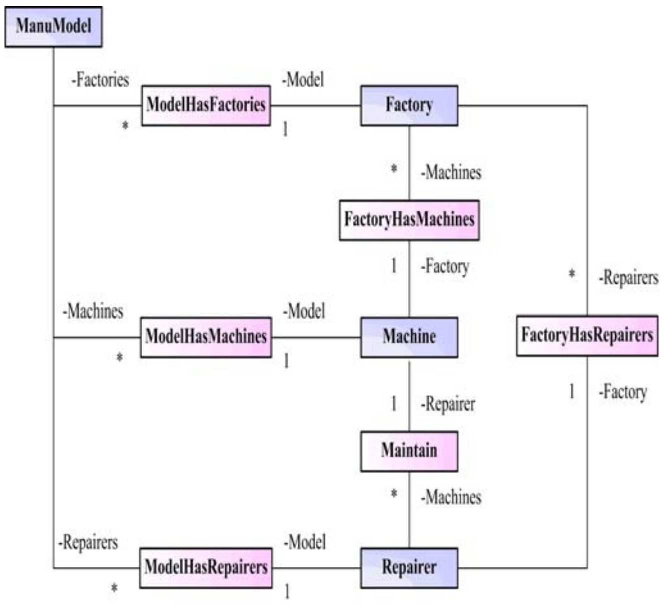
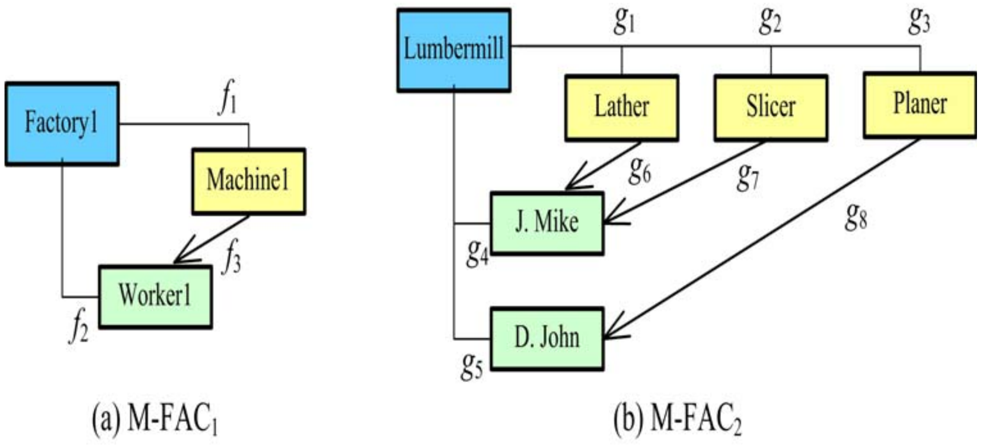
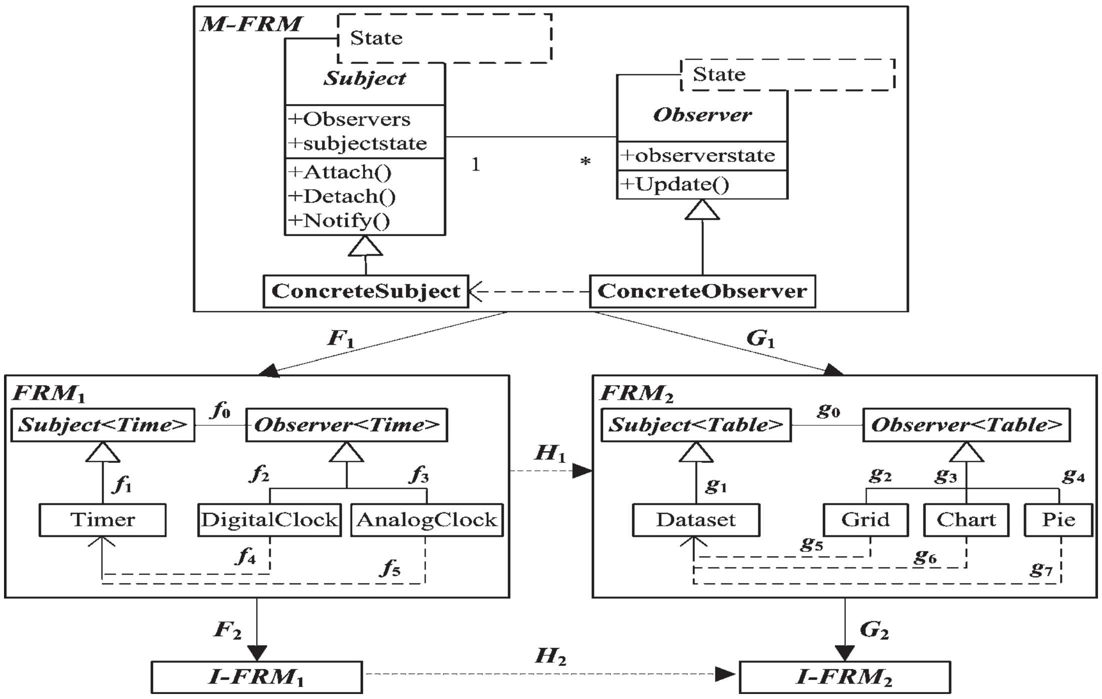


Figure 8. Domain-specific language “TheManufactory”





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*$_2$ that contains the following four parts:

- $f_0 \rightarrow g_0$
- $f_1 \rightarrow g1$
- *equalizer*(f_2; f_3) $\rightarrow$ *equalizer*(g_2; g_3; g_4)
- *equalizer*(f_4; f_5) $\rightarrow$ *equalizer*(g_5; g_6; g_7)

Thereby, the new implemented framework *I-FRM*$_2$ is generated without refining$G_2 o G_1$ manually: G_1 is the composition H_1 *o* F_1, while G_2 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. 2003CCA02800) 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

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

[2] Abdullah, M.S., Kimble, C., Paige, R. Benest, I., Evans, A. (2005). Developing a UML Profile for Modelling Knowledge-Based Systems. Lecture Notes in Computer Science, vol. 3599, 220-233.

[3] Alexandrescu, A (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Reading MA: Addison-Wesley.

[4] Allen, B.P.,Lee, S.D (1989). A Knowledge-based Environment for the Development of Software Parts Composition Systems, *In*: Proceedings of 11th International Conference on Software Engineering, Pittsburgh, PA. 104-112.

[5] Asperti, A., Longo, G (1991). Categories, Types and Structures: an introduction to category theory for the working computer scientist. Cambridge: MIT Press. 306.

[6] Bidoit, M., Sannella, D., Tarlecki, A (1998). Architectural Specifications in CASL, *In*: Proceedings of 7th International Conference on Methodology and Software Technology, Lecture Notes in Computer Science, vol. 1548, 341-357.

[7] Blaine, L., Gilham, L.M., Goldberg, A., Jullig, R., McDonald, J., Srinivas, Y.V. (Ed.) (1994). Slang Language Manual: Specware Version Core4. Kestrel Institute.

[8] DeLoach, S.A.,Hartrum, T.C. (2000). A Theory-Based Representation for Object-Oriented Domain Models. IEEE Transactions on Software Engineering, 26 (6) 500-517.

[9] Devanbu, P., Brachman, R.J, et al. (1990). LaSSIE: A Knowledge-based Software Information System, *In*: Proceedings of 12th International Conference on Software Engineering, Nice, France. 249-261.

[10] Ehrich, H.D.,Gogolla, M. (1991). Objects and Their Specifications. In Proceedings of 8th Workshop on Abstract Data Types. Lecture Notes in Computer Science, vol. 665, 40-65.

[11] Fiadeiro, J., Maibaum, T. (1991). Describing, Structuring and Implementing Objects. Rex90 Workshop on the Foundations of Object Oriented Languages, Lecture Notes in Computer Science, vol. 489, 274-310.

[12] Gamma, E., Helm, R., Johnson, R., Vlissides J. (1995). Design Patterns: Elements of Reusable Object-Oriented Systems. Reading MA: Addison-Wesley. 395.

[13] Goguen, J.A (1991). A Categorical Manifesto. Mathematical Structures in Computer Sciences, 1(1), 49-67.

[14] Henninger, S. (1994). Using Iterative Refinement to Find Reusable Software, IEEE Software, 11 (5) 48-59.

[15] Kennedy, G.J. (2004). Design Patterns in Information System Development, *In*: Proceeding of 5th Australasian Workshop on Software and System Architectures, Melbourne, Australia, 12-18.

[16] Kurtev, I., Berg, K., Aksit, M (2003). UML to XML-Schema Transformation: a Case Study in Managing Alternative Model Transformations in MDA, *In*: Proceedings of the Forum on Specification and Design Languages, Frankfurt, Germany.

[17] Lu, X.M., Dillon, T.S. (1994). An Algebraic Theory of Object-Oriented Systems. IEEE Transactions on Knowledge and Data Engineering, 6 (3) 412-419.

[18] Mi, P.W., Lee, M.J., Scacchi, W (1992). A Knowledge-Based Software Process Library for Process-Driven Software Development. Proceedings of IEEE 7th Conference on Knowledge-Based Software Engineering, Washington. 121-132.

[19] Microsoft Corp.: Visual Studio SDK September 2006 Help. Avariable at: http://msdn.microsoft.com/vstudio/dsltools/, 2006-8-18.

[20] Mylopoulos, J., Borgida. A., Yu, E. (1997). Representing Software Engineering Knowledge. Automated Software Engineering, 4 (3) 291-317.

[21] Saunders, M.L (1998). Categories for the Working Mathematician. New York: Springer-Verlag. 314.

[22] Smith, D.R. (1999). Designware: Software Development by Refinement. Proceedings of 8th International Conference on Category Theory and Computer Science, The Kluwer International Series In Engineering And Computer Science. 3-21.

[23] Spivey, J.M (1989). The Z Notation: A Reference Manual. New York: Prentice Hall.

[24] Wood, M., Sommerville, I. (1998). A Knowledge-based Software Components Catalogue. Software Engineering Environments (ed. Brereton, P.). Ellis Horwood Limited. 116-133.

[25] Wang, J.Q., Wang, K., Zheng, Y.J. (2004). The Application of CBD Approach in Materiel Support Information Management System. Journal of Academy of Armored Force Engineering, 18 (4) 58-61, 70.

[26] Wiels, V., Easterbrook, S (1998). Management of Evolving Specifications Using Category Theory. Proceedings of IEEE 13th International Conference on Automated Software Engineering, Hawaii. 12-21.

[27] Zheng, Y.J. and Xue, J.Y (2005). MISCE: A Semi-Automatic Development Environment for Logistic Information Systems. Proceedings of IEEE 1st International Conference on Service Operations and Logistics, and Informatics, Beijing, China. 1020-1025.

[28] Zheng, Y.J., Xue, J.Y., Liu, W.B (2006). Object-Oriented Specification Composition and Refinement via Category Theoretic Computations. Proceedings of 3rd International Conference on Theory and Applications of Models of Computation. Lecture Notes in Computer Science, vol. 3959, 601-610.

[29] Zheng, Y.J., Shi, H.H., Xue, J.Y (2006). Formalization and Mechanization of Design Patterns. Proceedings of 1st International Conference on Computer Science & Education, Xiamen, China, 2006. 892-897.

[30] Zheng, Y.J., Wang, J.Q., Xue, J.Y (2006). Developing Reliable Software of Logistic Information System with SPEC#. Computer Engineering and Design, 27 (22) 4178-4182.