Design and Implementation of DSL via Category Theoretic Computations

Haifeng Ling, Xianzhong Zhou
School of Management & Engineering, Nanjing Univ
Nanjing Jiangsu 210093, China
ling_11178@sina.com

Yujun Zheng
Institute of Software, CAS
Beijing 100080, China
yujun.zheng@computer.org

***Abstract*—Domain specific languages provide appropriate built-in abstractions and notations in a particular problem domain, and have been suggested as means for developing highly adaptable software systems. The paper presents a theory-based framework to support domain-specific design and implementation. Focusing concern on reasoning about interactive relationships among software models and objects at different levels of abstraction and granularity, our framework provides a unified categorial environment for intra-model composition and inter-model refinement of specifications via category theoretic computations, and therefore enables a high-level of reusability and dynamic adaptability.**

I. Introduction

Object-orientation is recognized as an important advance in software technology, particularly in modeling complex phenomena more easily than its predecessors [1]. But the progress in reusability, maintainability, reliability, and even expressiveness has fallen short of expectations. Current research and practical experience suggest that achieving significant progress with respect to software reuse requires a paradigm shift towards modeling and developing software system families rather than individual systems [2].

Domain-specific languages (DSLs) are such an approach that focuses on the effective specification, implementation, and verification of system-family programs in a particular problem domain, and thus have been suggested as means for developing robust and reliable software systems [3]. They have aroused considerable interest from both academics and practitioners, and recent efforts (e.g. [4], [5], [6]) have particularly concentrated on the underlying infrastructure, supporting tools and integrated development environments for DSL modeling and implementation.

Being highly declarative and hiding much of the implementation details, DSLs can be considered more as specification languages than programming languages [7]. Recently, some researches have been contributed to improve the formalization of representation, composition, and validation for the kernel constructs of DSL specifications. In [8] Ledeczi advocated a UML-based metamodeling technique to rapid specification and development of domain-specific modeling languages, a key feature of which is the composition of new metamodels from existing ones through the use of newly defined UML operators including equivalence, implementation inheritance, and interface inheritance. Also using UML as a front-end for formal descriptions of DSL designs, Felfernig *et al* [9] focused on translating domain-specific knowledge into logical sentences and exploited by a general inference engine solving the configuration task automatically. To support advanced separation of concerns, Gray [10] applied the concepts of aspect-oriented programming to domain-specific modeling and constructed support tools that facilitate the elevation of crosscutting modeling concerns to first-class citizens. Nevertheless, there still lacks a unified approach to capturing formal semantics of collaboration relationships and responsibility distribution among inter-model and/or intra-model DSL specifications, which are recognized as having a significant impact on structuring and reasoning about the domain-specific software architectures.

As a "theory of functions", category theory offers a highly formalized language for software specifications [11], and is especially suited for focusing concern on reasoning about inter- and intra- relations between software objects. Also, it is sufficiently abstract that it can be applied to a wide range of different specification languages [12]. In this paper we extend the algebraic model of object-orientation to a theory-based framework that explicitly defines formal notions of domain-specific models at different levels of granularity and abstraction, which support mechanizable specification composition, refinement, and code generation via category theoretic computations.

In Section II we present some preliminaries of category theory, and then describe the construction of object- and model-oriented categories for domain-specific design and implementation in Section III. Section IV illustrates its application with case studies, and Section V concludes.

II. Category Theory

Definition 1 A **category** C is

- a collection Ob_C of objects
- a collection Mor_C of morphisms (arrows)
- two operations dom, cod assigning to each arrow f two objects (the domain and codomain of f)
- an operation id (identity) assigning to each object b a morphism id_b such that $dom(id_b) = cod(id_b) = b$

978-0-7695-3932-4/09 $26.00 © 2009 IEEE
DOI 10.1109/FCST.2009.102

- an operation $\circ$ (composition) assigning to each pair f, g of arrows with $dom(f) = cod(g)$ and arrow $f \circ g$ such that $dom(f \circ g) = dom(g)$, $cod(f \circ g) = cod(f)$
- identity and composition must satisfy: (1) for any arrows f, g such that $cod(f) = b = dom(g)$, we have $id_b \circ f = f$ and $g \circ id_b = g$; (2) for any arrows f, g, h such that $dom(f) = cod(g)$ and $dom(g) = cod(h)$, we have $(f \circ g) \circ h = f \circ (g \circ h)$

Definition 2 A **diagram** D in a category C is a directed graph whose vertices $i \in I$ are labeled by objects $d_i \in Ob_C$ and whose edges $e \in E$ labeled by morphisms $f_e \in Mor_C$.

Definition 3 Let D be a diagram in C, a **cocone** to D is
- a C-object x
- a family of morphisms $\{f_i: d_i \to x | i \in I\}$ such that for each arrow g: $d_i \to d_j$ in D, we have $f_j \circ g = f_i$, as shown in Figure 1(a).

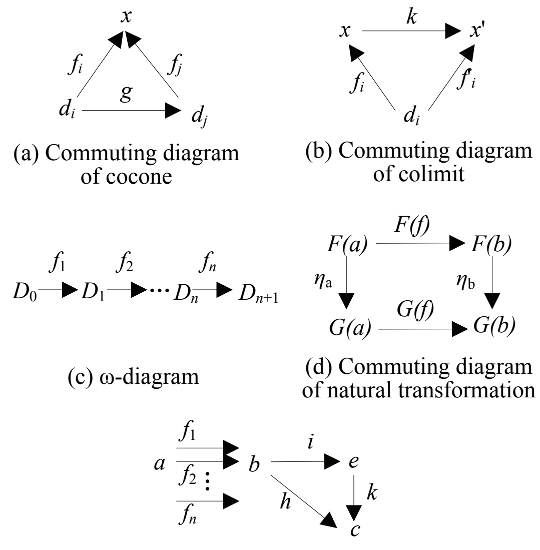
Definition 4 Let D be a diagram in C, a **colimit** to D is a cocone with C-object x such that for any other cocone with C-object x', there is a unique arrow k: $x \to x'$ in C such that for each d_i in D, f_i: $d_i \to x$ and f'_i: $d_i \to x'$, we have $k \circ f_i = f'_i$, as shown in Figure 1(b).

Definition 5 Let C and D be categories, a **functor** F: $C \to D$ is a pair of operations F_{ob}: $Ob_C \to Ob_D$, F_{mor}: $Mor_C \to Mor_D$ such that, for each morphism f: $a \to b$, g: $c \to d$ in C,
- $F_{mor}(f)$: $F_{ob}(a) \to F_{ob}(b)$
- $F_{mor}(f \circ g) = F_{mor}(f) \circ F_{mor}(g)$
- $F_{mor}(id_a) = id_{F_{ob}(a)}$

Definition 6 An ω**-diagram** in a category C is a diagram with the structure shown in Figure 1(c), and
- a category C is ω**-cocomplete** if and only if it has colimits for all ω-diagrams.
- a functor F is ω**-continuous** if and only if it preserves all colimits for ω-diagrams.

Definition 7 Let C and D be categories and F, G: $C \to D$ be functors, a **natural transformation** η: $F \to G$ is a function that assigns each object a in C a morphism η_a : $F_{ob}(a) \to G_{ob}(a)$ in D, such that for each f: $a \to b$ in C we have $\eta_b \circ F_{mor}(f) = G_{mor}(f) \circ \eta_a$ (see Figure 1(d)).

Definition 8 Given a family of morphisms $f_1, f_2 \dots f_n \in Mor[a, b]$, an **coequalizer** of them is an object e and a morphism $i \in Mor[b, e]$ such that (1)$i \circ f_1 = i \circ f_2 = \dots = i \circ f_n$; (2) for each $h \in Mor[b, c]$, $h \circ f_1 = h \circ f_2 = \dots = h \circ f_n$, there is an $k \in Mor[e, c]$ such that $k \circ i = h$ (Figure 1(e)).



$$\begin{tikzcd} & x & \\ d_i \arrow[ur, "f_i"] \arrow[rr, "g"'] & & d_j \arrow[ul, "f_j"'] \end{tikzcd}$$

(a) Commuting diagram of cocone

$$\begin{tikzcd} x \arrow[rr, "k"] & & x' \\ & d_i \arrow[ul, "f_i"] \arrow[ur, "f'_i"'] & \end{tikzcd}$$

(b) Commuting diagram of colimit

$$\begin{tikzcd} D_0 \arrow[r, "f_1"] & D_1 \arrow[r, "f_2"] & \cdots D_n \arrow[r, "f_n"] & D_{n+1} \end{tikzcd}$$

(c) ω-diagram

$$\begin{tikzcd} F(a) \arrow[r, "F(f)"] \arrow[d, "\eta_a"'] & F(b) \arrow[d, "\eta_b"] \\ G(a) \arrow[r, "G(f)"] & G(b) \end{tikzcd}$$

(d) Commuting diagram of natural transformation

$$\begin{tikzcd} a \arrow[r, shift left=2, "f_1"] \arrow[r, "f_2 \vdots"'] \arrow[r, shift right=2, "f_n"'] & b \arrow[r, "i"] \arrow[dr, "h"'] & e \arrow[d, "k"] \\ & & c \end{tikzcd}$$

(e) Commuting diagram of coequalizer

Figure 1. Basic diagrams in category theory

Definition 9 A **signature** $\Sigma =< S, \Omega >$, where S denotes a set of sort symbols, and Ω denotes a set of operators. In more detail, $\Omega =< C, F, P >$, 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 10 ***SIG*** is a category with signatures as objects, and a signature morphism is a consistent mapping from one signature to another.

Definition 11 A **specification** $SP =< \Sigma, \Phi >$, where Σ is a signature, and Φ is a (finite) set of axioms over Σ.

Definition 12 ***SPEC*** is a category with specifications as objects, and a specification morphism between specification $< \Sigma_1, \Phi_1 >$ and specification $< \Sigma_2, \Phi_2 >$ 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.

III. Modular Constructions

A. Category of Software Specifications

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 where a specification is a finite theory presentation.

B. Categories of Object-Oriented Specifications

As mentioned, 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 similar to the concepts in object-orientation. Next we show how to use the notions of category theory to abstract object-oriented design at different levels (which is an extensions of [18]):

Definition 13 An **object signature** $\theta =< \Sigma, A, \Gamma >$, where $\Sigma =< S, \Omega >$ is a (universe) signature, A is an $S^* \times S$-indexed family of attribute symbols, and Γ is an S^*-indexed family of action symbols.

Definition 14 ***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 15 An **object specification** $OSP =< \theta, \Phi >$, where θ is an object signature, and Φ is a set of θ-axioms.

Definition 16 ***OBJ-SPEC*** is a category with object specifications as objects, and a morphism between specification $< \theta_1, \Phi_1 >$ and specification $< \theta_2, \Phi_2 >$ is a mapping of signature θ_1 into signature θ_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 [15]. Here we construct the category of classes out of the category of objects in the second sense, and so are that of meta-classes and that of meta-meta-classes.

Definition 17 Let D_1, $D_2 \ldots D_n$ be ω-diagrams in $OBJ\text{-}SPEC$ 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 $OBJ\text{-}SPEC$ that between an object in D_1 and an object in D_2.

Definition 18 Let D_1, $D_2 \ldots D_n$ be ω-diagrams in $CLS\text{-}SPEC$ 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 $CLS\text{-}SPEC$ that between a class in D_1 and a class in D_2.

Definition 19 Let D_1, $D_2 \ldots D_n$ be ω-diagrams in $M\text{-}CLS\text{-}SPEC$ 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 $M\text{-}CLS\text{-}SPEC$ that between a meta-class in D_1 and a meta-class in D_2.

Functors from $CLS\text{-}SPEC$ to $OBJ\text{-}SPEC$ 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 $M\text{-}CLS\text{-}SPEC$ to $CLS\text{-}SPEC$ are the refinements from meta-classes to classes, and functors from $MM\text{-}CLS\text{-}SPEC$ to $M\text{-}CLS\text{-}SPEC$ the refinements from meta-meta-classes to meta-classes. Figure 2 illustrates the bottom-up way of object-oriented categories construction.

C. Categories of Model-Oriented Specifications

Any DSL that you define is referred to as a domain model [6], which can be viewed as the composition of individual domain classes while all the properties (relationships) are preserved. In terms of categorical models:

- Each DSL is a meta-model in the domain, which has domain classes representing meta-classes and domain relationships representing meta-class morphisms;
- Each model that abstracts a real-world system in the specific domain is an instance of the DSL, which has model classes and their morphisms;
- As a refinement of the system model, an executable system contains runtime, interactive objects as instances of model classes and morphisms;
- Rather than from scratch, most DSLs are designed within a higher-order and typed programming language [16], which can be treated as a meta-meta-model that contains meta-meta-classes and their morphisms.

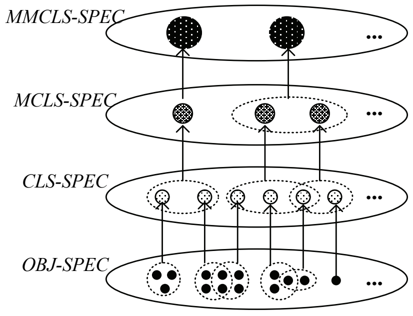


Figure 2. Bottom-up constructions of object-oriented categories

Therefore, it is straightforward to construct new categories of domain models out of $MM\text{-}CLS\text{-}SPEC$, $M\text{-}CLS\text{-}SPEC$, $CLS\text{-}SPEC$, and $OBJ\text{-}SPEC$ by composing their objects and relations (morphisms). **Definition 20** ***MM-MOD-SPEC*** is a category with diagrams in $MM\text{-}CLS\text{-}SPEC$ as objects, and a morphism between two meta-meta-models, namely $MMMOD_1$ and $MMMOD_2$, is the colimit of morphisms in $MM\text{-}CLS\text{-}SPEC$ that between a meta-meta-class of $MMMOD_1$ and a meta-meta-class of $MMMOD_2$.

Definition 21 ***M-MOD-SPEC*** is a category with diagrams in $M\text{-}CLS\text{-}SPEC$ as objects, and a morphism between two meta-models, namely $MMOD_1$ and $MMOD_2$, is the colimit of morphisms in $M\text{-}CLS\text{-}SPEC$ that between a meta-class of $MMOD_1$ and a meta-class of $MMOD_2$.

Definition 22 ***MOD-SPEC*** is a category with diagrams in $CLS\text{-}SPEC$ as objects, and a morphism between two models, namely MOD_1 and MOD_2, is the colimit of morphisms in $CLS\text{-}SPEC$ that between a class of MOD_1 and a class of MOD_2.

Definition 23 ***I-MOD-SPEC*** is a category with diagrams in $OBJ\text{-}SPEC$ as objects, and a morphism between two model instances, namely $IMOD_1$ and $IMOD_2$, is the colimit of morphisms in $OBJ\text{-}SPEC$ that between an object of MOD_1 and an object of MOD_2.

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

IV. Implementations

A. Fundamental Approaches

Under this framework, category theoretic computations can be applied to support automatic reuse of software

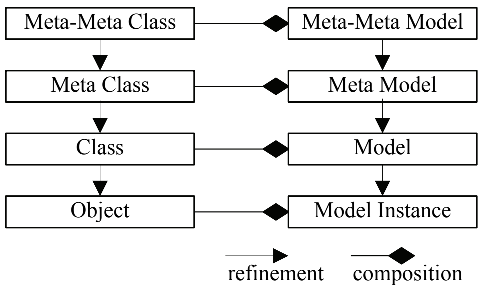


Figure 3. Horizontal constructions of model-oriented categories

designs and refinements. First, the basic principle to specify a system is to build and refine the specification for each component separately, and then use the colimit operation to compose these specifications and refinements. For two specifications (namely A_1 and B_1), the morphisms between them form a new specification (R_1) that expresses the relationships between the two components. By computing the colimit of f_1: $R_1 \to A_1$ and f_2: $R_1 \to B_1$, the composed specification (S_1) is automatically worked out.

Afterward, the colimit operation is applied to the existing refinements of individual specifications to generate a new refinement of the composed specification. That is, having refinements h_1: $A_1 \to A_2$ and h_2: $B_1 \to B_2$, instead of manually constructing a refinement $S_1 \to S_2$, we obtain S_2 by computing the colimit of f_1': $R_2 \to A_2$ and f_2': $R_2 \to B_2$, which are calculated by the composition $h_1 \circ f_1 \circ e^{-1}$ and $h_2 \circ f_2 \circ e^{-1}$ respectively, as illustrated in Figure 4(a).

The colimit operation can also be used for constructing refinements. Suppose a new specification S_1' has at least the same structure of an existing specification S_1 [17], instead of constructing the new refinement g, S_2' can be obtained by computing the colimit of e and f (see Figure 4(b)).

Furthermore, when specifications S_1 and S_1' refine a same specification (namely S_0) and refinements f: $S_0 \to S_1$ and g: $S_0 \to S_1'$ are both epimorphisms, S_1' itself can be worked out by constructing a functor F: $f \to g$ (see Figure 4(c)). That is, g can be constructed exclusively from f and F because e: $S_1 \to S_1'$ is an isomorphism, which is proved as follows: For morphisms id_1: $S_1 \to S_1$, id_2: $S_1' \to S_1'$, e_1: $S_1 \to S_1'$ and e_2: $S_1' \to S_1$, we have: (1) $id_1 \circ f = f = e_2 \circ g = e_2 \circ e_1 \circ f$; (2) $id_2 \circ g = g = e_1 \circ f = e_1 \circ e_2 \circ g$. Since f and g are epic and therefore right cancelable, we have $e_2 \circ e_1 = id_1$ and $e_1 \circ e_2 = id_2$, and thus e_1 is an isomorphism (see Figure 4(d)).

B. Constructing Model Instances

In most cases, a full, provably correct implementation of a model into a model instance (executable system) can be treated as an epimorphism, and we can get other model instances by constructing a functor from the existing implementation. E.g., Figure 5(a) gives a system model

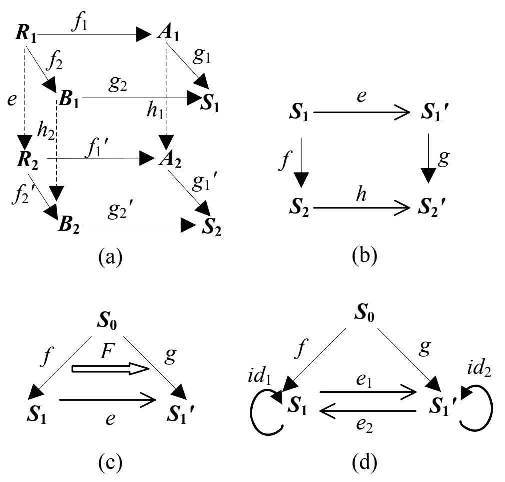


Figure 4. Specification constructions via category theoretic computations

named $M\text{-}FAC_1$ that refines each meta-class and meta-relationship of a simple DSL $TheManufactory$ (partly shown in Figure 6(a)) into exact one concrete class and relationship respectively; A model instance named $IM\text{-}FAC_1$, is created by building class instances into generic lists and relationship instances into class properties. The kernel code fragment of $IM\text{-}FAC_1$ 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);
```

Such code is generated based on the following refinement:

```
Golbal.Factories = new List<Factory>;
<#foreach (Factory fac1 in ManuModel.Factories){#>
fac1 = new Factory("<#= fac1.Name #>");
fac1.Workers = new List<Worker>;
fac1.Machines = new List<Machine>;
 <#foreach (Worker wor1 in fac1.Workers){#>
 wor1 = new Worker("<#= wor1.Name #>");
 fac1.Workers.Add(wor1);
  <#foreach (Machine mac1 in wor1.Machines)
  {#>
  mac1 = new Machine("<#= mac1.Name #>");
  wor1.Machines.Add(mac1);
  fac1.Machines.Add(mac1);
  <#}#>
 <#}#>
<#}#>
```

Here we want to generate another model instance of database program. By constructing a functor F: $f \to g$ that transforms each list of a class into a major data table and each list of a relationship into a cross data table, we get the new refinement g as follows:

```
Golbal.FactoryTable = new DbTable();
Golbal.WorkerTable = new DbTable();
Golbal.MachineTable = new DbTable();
Golbal.FactoryWorkerTable = new DbTable();
Golbal.WorkerMachineTable = new DbTable();
Golbal.FactoryMachineTable = new DbTable();
<#foreach (Factory fac1 in ManuModel.Factories){#>
Golbal.FactoryTable.Insert(nID,"<#=fac1.Name#>");
 <#foreach (Worker wor1 in fac1.Workers){#>
 Golbal.WorkerTable.Insert(nID, "<#=
 wor1.Name #>");
 Golbal.FactoryWorkerTable.Insert(nID,
 "<#=fac1.Name#>", "<#= wor1.Name #>);
  <#foreach (Machine mac1 in wor1.Machines)
  {#>
  Golbal.WorkerTable.Insert(nID,("<#=
  mac1.Name #>");
  Golbal.WorkerMachineTable.Insert(nID,
  "<#=wor1.Name#>", "<#= mac1.Name #>);
  Golbal.FactoryMachineTable.Insert(nID,
  "<#=fac1.Name#>", "<#= mac1.Name #>);
  <#}#>
 <#}#>
<#}#>
```

And the code fragment of the new model instance generated by g over $M\text{-}FAC_1$ is:

```
Golbal.FactoryTable = new DbTable();
Golbal.WorkerTable = new DbTable();
Golbal.MachineTable = new DbTable();
Golbal.FactoryWorkerTable = new DbTable();
Golbal.WorkerMachineTable = new DbTable();
Golbal.FactoryMachineTable = new DbTable();
Golbal.FactoryTable.Insert(nID, "Factory1");
Golbal.WorkerTable.Insert(nID, "Worker1");
Global.FactoryWorkerTable.Insert(nID, "Factory1",
 "Worker1");
Golbal.MachineTable.Insert(nID, "Machine1");
Global.WorkerMachineTable.Insert(nID, "Worker1",
 "Machine1");
Global.FactoryMachineTable.Insert(nID, "Worker1",
 "Machine1");
```

C. Constructing Models

Base on a DSL, we can create a series of models, which are referred to as instances of the domain model. When creating a new model, if there is a morphism between it and an existing model that has been implemented into executable system(s), the morphism can be utilized to implement the new model mechanically. As illustrated in Figure 7(a), suppose the refinement path f_1: $DSL_1 \rightarrow MOD_1$ and f_2: $MOD_1 \rightarrow I\text{-}MOD_1$ already exist, by constructing h_1: $MOD_1 \rightarrow MOD_2$, the refinement path $g_2 \circ g_1$ can be worked out automatically using category theoretic computations. In detail, g_1 is the composite morphism $h_1 \circ f_1$, while g_2 and h_2 can be obtained by computing the colimit of h_1 and f_2. In such a way we eliminate the requirement for refining g_1 and g_2 manually, and the target system $I\text{-}MOD_2$ is generated without repetitious efforts and potential error introduction.

Now suppose the meta-model $TheManufactory$ in Figure 6(a) need to be refined into a new model, namely $M\text{-}FAC_2$, which has three instances of meta-class $Machine$ and two instances of meta-class $Worker$, as described in Figure 5(b). Here an inter-model morphism (namely h) does exist between the $M\text{-}FAC_1$ and $M\text{-}FAC_2$, which consists of the following three parts (intra-model morphisms are labeled in Figure 5):

- $f_1 \rightarrow coequalizer(g_1, g_2, g_3)$
- $f_2 \rightarrow coequalizer(g_4, g_5)$
- $f_3 \rightarrow coequalizer(coequalizer(g_6, g_7), g_8)$

Based on the model morphism together with the existing implementation of $M\text{-}FAC_1$, implementation of $M\text{-}FAC_2$ can be worked out automatically. In this example, the colimit of h and f results in a model instance of $M\text{-}FAC_2$ 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);
```

D. Constructing Domain Models

The model refinement process can also be scaled up to the level of meta-model. That is, if two meta-models (DSLs) refine a same meta-meta model, and there is a morphism between the meta-models, the refinement path of one meta-model can be applied to another directly, as illustrated in Figure 7(b).

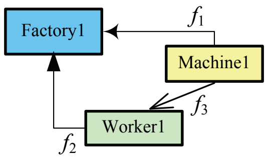
For example, starting from the domain model $TheManufactory$, we can generate a new DSL $TheTransporter$ by constructing a morphism that consists of the following parts:

- *ManuModel* → *TranModel(root)*
- (*ManuModel* → *Factory*) → (*TranModel* → *Company*)
- (*ManuModel* → *Machine*) → (*TranModel* → *Truck*)
- (*ManuModel* → *Worker*) → (*TranModel* → *Driver*)
- (*Factory* → *Worker*) → (*Company* → *Driver*)
- (*Factory* → *Machine*) → (*Company* → *Truck*)
- (*Worker* → *Machine*) → (*Driver* → *Truck*)

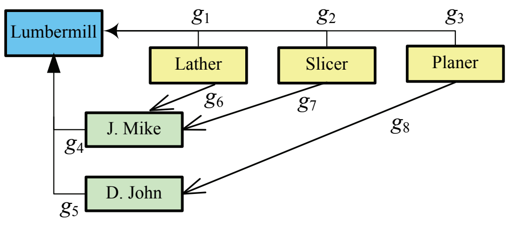
The result domain model is partly shown in Figure 6(b), and hence the refinements of $TheManufactory$ in the above two subsections can be applied to $TheTransporter$ directly and mechanically.

V. Conclusion

DSLs provide appropriate built-in abstractions and notations in a particular problem domain, and have been suggested as means for developing highly adaptable and reliable software systems. By being combined with the



(a) M-FAC$_1$



(b) M-FAC$_2$

Figure 5. Sample models of the "TheManufactory" domain

formal, mathematically precise methods, DSLs can significantly enhance their capability to automatically structure and reason about the software architectures.

The paper presents a theory-based framework in which model-oriented specifications are constructed by composing individual object-oriented specifications while all the properties are preserved, and the model refinements can be automatically generated by the computation of existing morphisms. The major benefit of our approach is twofold: First, it explicitly provides a systemic, formal semantics for domain-specific models, and therefore enables a high-level of reusability and dynamic adaptability. Second, it utilizes the ability of categorical computations to support automated composition, refinement, and validation for DSL specifications at different levels of abstraction and granularity. Our ongoing efforts include building a knowledge base for software architectural design in specific domains such as logistics, manufacturing, and disaster management [18].

References

[1] Meyer, B.: Object-Oriented Software Construction (2nd ed.). Prentice Hall, New York, 1997.

[2] Czarnecki, K.: Overview of Generative Software Development. In Proc. Unconventional Programming Paradigms, Mont Saint-Michel, France, Lecture Notes in Computer Science 3566, pp. 313-328, 2005.

[3] Gupta, G.: A Language-centric Approach to Software Engineering: Domain Specific Languages meet Software Components. In Proc. CologNet Workshop on Logic Programming, 2002, Madrid, Spain.

[4] Batory, D. Lofaso, B., and Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In Proc. 5th Int'l Conf. Software Reuse, Victoria, Canada, 1998, pp. 143-153.

[5] Nordstrom, G.G.: Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments. In Proc. IEEE Conf. & Worshop on Engineering of Computer-Based Systems, Nashville, Tennessee, 1999, pp. 68-74.

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

[7] Frechot, J.: Domain-Specific Languages-An Overview. Available at: http:// compose.labri.fr/documentation/dsl, 2003-09.

[8] Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., and Maroti, M.: On Metamodel Composition. In Proc. IEEE Int'l Conf. Control Applications, Mexico City, Mexico, 2001, pp. 756-760.

[9] Felfernig, A., Friedrich, G., and Jannach, D.: UML As Domain Specific Language For The Construction Of Knowledge-Based Configuration Systems. International Journal of Software Engineering and Knowledge Engineering, vol. 10, no. 4, pp. 449-469, 2000.

[10] Gray, J., Bapty, T., Neema, S., and Gokhale, A.: Aspect-Oriented Domain-Specific Modeling. Communications of the ACM, vol. 44, no. 10, pp. 87-93, 2001.

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

[12] Wiels, V. and Easterbrook, S.: Management of Evolving Specifications Using Category Theory. In Proc. 13th IEEE Conf. Automated Software Engineering, Hawaii, pp. 12-21, 1998.

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

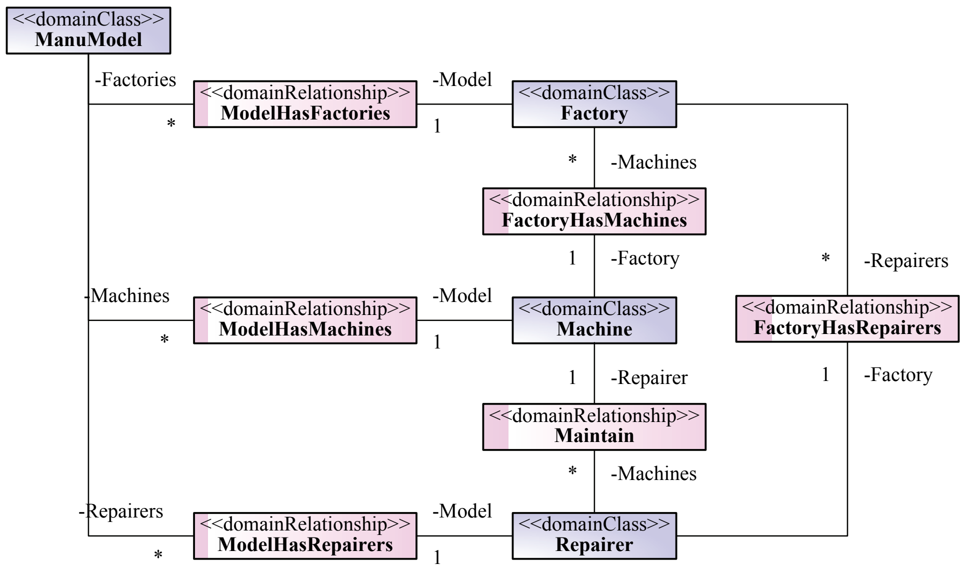
[14] Zheng, Y.J., Xue, J.Y., and Liu,W.B.: Object-Oriented Specification Composition and Refinement via Category Theoretic Computations. In Proc. 3rd Int'l Conf. Theory and Applications of Models of Computation, Beijing, China, Lecture Notes in Computer Science 3959, pp. 601-610, 2006.

[15] Ehrich, H.D. and Gogolla, M.: Objects and Their Specifications. In Proc. 8th Workshop on Abstract Data Types. Lecture Notes in Computer Science 665, pp. 40-65, 1991.

[16] Hudak, P.: Modular domain specific languages and tools. In Proc. 5th Int'l Conf. Software Reuse, Victoria, Canada, 1998, pp. 134-142.

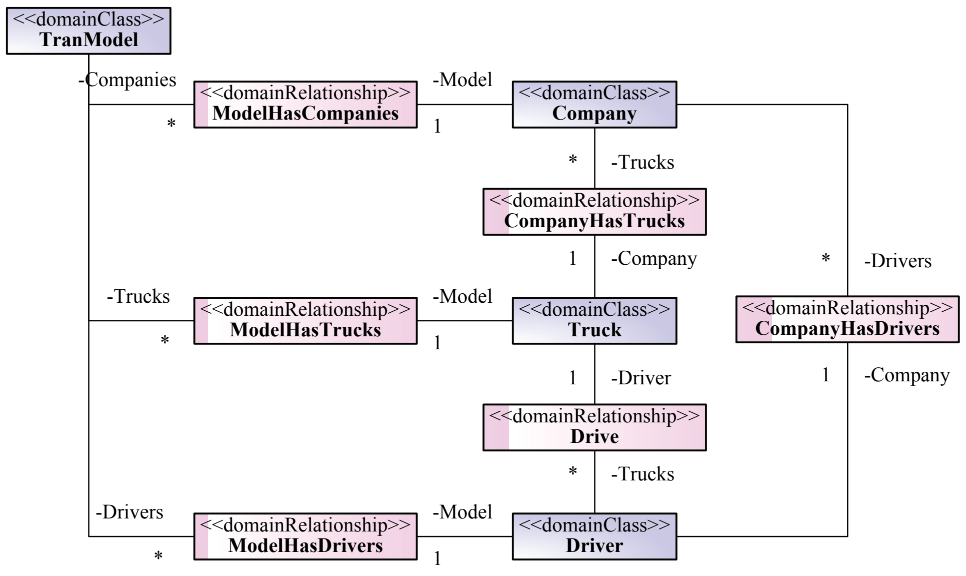
[17] Smith, D.R.: Designware: Software Development by Refinement. In Proc. 8th Conf. Category Theory and Computer Science. The Kluwer International Series in Engineering & Computer Science, pp. 3-21, 1999.

[18] Zheng, Y.J. and Xue, J.Y.: Knowledge-Based Support for Object-Oriented Software Design and Synthesis: a category theoretic approach. In Proc. IEEE 6th Int'l Conf. Intelligent System Design and Applications, Jinan, China, 2006, vol. 1, pp. 759-764.



<<domainClass>> ManuModel
-Factories
*
<<domainRelationship>> ModelHasFactories
-Model
1
<<domainClass>> Factory
*
-Machines
<<domainRelationship>> FactoryHasMachines
1
-Factory
*
-Repairers
-Machines
*
<<domainRelationship>> ModelHasMachines
-Model
1
<<domainClass>> Machine
<<domainRelationship>> FactoryHasRepairers
1
-Repairer
1
-Factory
<<domainRelationship>> Maintain
*
-Machines
-Repairers
*
<<domainRelationship>> ModelHasRepairers
-Model
1
<<domainClass>> Repairer

(a) TheManufactory



<<domainClass>> TranModel
-Companies
*
<<domainRelationship>> ModelHasCompanies
-Model
1
<<domainClass>> Company
*
-Trucks
<<domainRelationship>> CompanyHasTrucks
1
-Company
*
-Drivers
-Trucks
*
<<domainRelationship>> ModelHasTrucks
-Model
1
<<domainClass>> Truck
<<domainRelationship>> CompanyHasDrivers
1
-Driver
1
-Company
<<domainRelationship>> Drive
*
-Trucks
-Drivers
*
<<domainRelationship>> ModelHasDrivers
-Model
1
<<domainClass>> Driver

(b) TheTransporter

Figure 6. Examples of domain models

$$\begin{tikzcd}
 & \text{DSL}_1 \arrow[dl, "f_1"'] \arrow[dr, dotted, "g_1"] & \\
\text{MOD}_1 \arrow[rr, "h_1"] \arrow[d, "f_2"'] & & \text{MOD}_2 \arrow[d, dotted, "g_2"] \\
\text{I-MOD}_1 \arrow[rr, "h_2"] & & \text{I-MOD}_2
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & \text{MM-MOD}_1 \arrow[dl, "f_0"'] \arrow[dr, dotted, "g_0"] & \\
\text{M-MOD}_1 \arrow[rr, dashed, "h_1"] \arrow[d, "f_1"'] & & \text{M-MOD}_2 \arrow[d, dotted, "g_1"] \\
\text{MOD}_1 \arrow[rr, dotted, "h_2"] \arrow[d, "f_2"'] & & \text{MOD}_2 \arrow[d, dotted, "g_2"] \\
\text{I-MOD}_1 \arrow[rr, dotted, "h_3"] & & \text{I-MOD}_2
\end{tikzcd}$$

(b)

Figure 7. Constructing model/domain model morphisms