Journal of Universal Computer Science, vol. 16, no. 17 (2010), 2455-2474
submitted: 15/2/10, accepted: 30/8/10, appeared: 1/9/10 © *J.UCS*

Verification of Structural Pattern Conformance Using Logic Programming

Lunjin Lu, Dae-Kyoo Kim, Yuanlin Zhu and Sangsig Kim
(Oakland University, Rochester, Michigan, USA
L2Lu,kim2,yzhu2,skim2345@oakland.edu)

Abstract: This paper formalizes UML class diagrams and structural patterns as mathematical objects and provides a precise notion of conformance of a structural model specified as a class diagram to a structural pattern. We also present a conformance verification method which represents a class diagram as a logic program and a structural pattern as a query. The conformance of the class diagram to the structural pattern is verified by computing all the answers to the query by the logic program and checking the satisfaction of realization multiplicity constraints imposed by the pattern.
Key Words: Design pattern, logic programming, pattern conformance, Prolog, UML
Category: D.2.10, D.2.13

1 Introduction

Software development can greatly benefit from reusing existing artifacts including architectural patterns, design patterns, design aspects, software components and code. Design patterns have been an important research subject in the area of software engineering, particularly in reuse-based software engineering since their introduction in computer science [Cunningham and Beck 1986]. A design pattern describes a proven solution based on the previous experience to a recurring design problem in a reusable form (e.g., see [Gamma et al. 1995]). By reusing high quality solutions, design patterns help the development of systems that are extensible, flexible and maintainable [Prechelt et al. 2002].

Evaluation of pattern conformance of designs is concerned with checking valid realizations of a pattern in a design within the context of the application being built. In general, realizing a pattern heavily relies on designer's experience and knowledge of the pattern. Invalid realization of a pattern, however, could deteriorate rather than improve quality of design. Therefore, a question that naturally arises is "how can one ensure validity of a pattern realization?" The question can be partly addressed by pattern formalization efforts (e.g., see [Eden 1999], [Guennec et al. 2000], [Kim 2004], [Lano et al. 1996], [Lauder and Kent 1998], [Mapelsden et al. 2002], [Mikkonen 1998] and [OMG 2003]) that facilitate pattern realization (instantiation). For example, template-based approaches (e.g., [OMG 2003]) formalize patterns in terms of parameters, and a pattern can be instantiated (realized) by stamping out the template. However, in many cases, instantiated pattern realizations require significant modifications such as adding new elements and modifying or removing some instantiated elements to accommodate application-specific requirements. Since these activities may break pattern conformance and compromise the benefits of using design patterns, pattern conformance must be checked.

There has been much work on identifying pattern instances at code level [Albin-Amiot et al. 2001], [Antoniol et al. 1998], [Balanyi et al. 2003], [Brown 1996], [Fabry & Mens 2004], [Heuzeroth et al. 2003], [Keller et al. 1999], [Philippow et al. 2003], [Shull et al. 1996] where structural properties (e.g., operations, attributes, relationships) of design patterns [Gamma et al. 1995] are searched for in code. These works support the reverse engineering efforts at the programming level so as to understand legacy systems and improve their quality attributes. However, there is little work on validating pattern instances at the model level which can greatly improve the quality of design and reduce development cost by finding errors in early development phase. Based on our study, we found that some of the programming-level work (e.g., [Brown 1996, Keller et al. 1999, Philippow et al. 2003]) can be extended for detecting model-level pattern instances. However, a significant limitation in this work is that a pattern specification represents a typical instance of design pattern. This limits its applicability because in most cases, a design pattern is realized in various forms depending on the application domain, and thus it is very rare to find exactly the same instance in different designs.

To address this issue, we formalize class diagrams [OMG 2003] and structural patterns as mathematical objects and provide a precise notion of conformance of a structural model specified as a class diagram to a structural pattern. We then use logic programming to rigorously verify conformance of a structural model to a structural pattern. We represent a structural pattern as a query and a structural model as a logic program. By utilizing inference capability of the logic programming language Prolog, we obtain an automated method that finds the instances of the pattern in the model.

The main contributions of this work are a precise notion of structural pattern conformance and the representation scheme in which a model is represented as logic programs and a pattern as a query. The representation scheme facilitates use of design patterns in software development as follows:

- The scheme can be used to find all instances of a pattern in a model by executing the program for the query. Each answer to the query is an instance of the pattern in the model. Thus, our approach does not simply tell if the model satisfies the pattern, but also informs "how" the model satisfies the pattern. Using a debugging technique [Shapiro 1982, Lu 2005], one can also identify the cause of non-conformance if the query does not have any answer.
- The scheme can also be used to validate a designer's assignment of pattern

roles to model elements. During the development of a software model, the designer may designate certain elements to play particular pattern roles. The scheme can be used to validate this assignment by representing it as an equality constraint and executing the constraint and the query with the program. Furthermore, given a partial mapping of pattern roles and model elements, the scheme can complete the mapping by logic inference.

The remainder of the paper is organized as follows. Section 2 formalizes class diagrams and structural patterns as mathematical objects and provides a precise notion of conformance. Section 3 presents how patterns and class diagrams can be represented in Prolog using the Visitor pattern and a price calculation application as examples. Section 4 gives an overview of related work, and Section 5 concludes the paper. This paper is an extended version of [Kim and Lu 2006].

2 Class Diagrams, Structural Patterns and Conformance

This section formalizes class diagrams and structural patterns as mathematical objects and then defines the notion of a class diagram conforming to a structural pattern.

2.1 Class diagrams

A class diagram consists of classifiers and relationships between these classifiers. We consider two kinds of relationships in this work: subtyping [1] and association [2] relationships. Subtyping relationship between classifiers is required to form a partial order. Each association has a name and a number of association ends. An association end is characterized by the the classifier at the end of the association, the position of the classifier in the association, its navigability and multiplicity. The multiplicity is an interval over natural numbers N with an added element `many` meaning unbounded. Extend $\leq$ over natural numbers by $i \leq \texttt{many}$ for all $i \in N$ and $\texttt{many} \leq \texttt{many}$. Let **Bound** $= \{L..U \mid L, U \in N \cup \{\texttt{many}\} \wedge L \leq U\}$ denote the set of intervals. We use Boolean values in **Bool** $= \{true, false\}$ to denote navigability of an association end. **Type** denotes the set of types that may be used in a class diagram including classifiers and primitive types.

We assume that the set of meta-classes **MetaClass** that may occur in a class diagram is given. Elements in **MetaClass** are ordered by a partial order $\preceq$ such that $k_1 \preceq k_2$ indicates that a class k_1 is a sub-meta-class k_2. For instance, a class can be refined to be a concrete, denoted $concrete \preceq class$. The partial

[1] A subtyping relationship occurs in a class diagram as either inheritance relationship or an implementation relationship.
[2] A dependency relationship is a special kind of association relationship.

order $\langle \mathbf{MetaClass}, \preceq \rangle$ is an input to our method. For example, we may have

$$\mathbf{MetaClass} = \left\{ \begin{array}{c} concrete, abstract, class, interface, classifier, \\ association, dependency, usage \end{array} \right\}$$

ordered by

$$\preceq = \left\{ \begin{array}{c} \langle abstract, class \rangle, \langle concrete, class \rangle, \langle class, classifier \rangle, \\ \langle interface, classifier \rangle, \langle usage, dependency \rangle, \langle dependency, association \rangle \end{array} \right\}^{\star}$$

where $r^{\star}$ is the reflexive and transitive closure of the relation r.

The following definition formalizes a class diagram as a mathematical object. In this definition, a subtyping relationship is explicitly represented in a partial order, whilst an association relationship is represented by its ends which are treated as features of participating classifiers. Let **Name** be a denumerable set of names for classifiers, relationships, attributes and operations.

Definition 1. A class diagram is a tuple $D = \langle \mathcal{C}, \sqsubseteq, \mathcal{R}, mclass, attr, oper, assoc \rangle$ where

- $\mathcal{C}$ is a set of classifiers such that $\langle \mathcal{C}, \sqsubseteq \rangle$ is a partial order;
- $\mathcal{R}$ is a set of relationships;
- $mclass : \mathcal{C} \cup \mathcal{R} \mapsto \mathbf{MetaClass}$ is a function that gives the meta-class of a classifier or a relationship;
- $attr : \mathcal{C} \mapsto \wp(\mathbf{Name} \times \mathbf{Type})$ is a function such that $attr(c)$ is a set of attributes each of which is characterized by its name and type. An interface does not have any attribute. We thus require $mclass(c) = interface \Rightarrow attr(c) = \emptyset$ where $\Rightarrow$ is the classical logical implication operator;
- $oper : \mathcal{C} \mapsto \wp(\mathbf{Name} \times \mathbf{Type}^{\star} \times \mathbf{Type})$ is a function such that $oper(c)$ is a set of operations. Each operation is characterized by its name and types of its arguments and the type of its return value;
- $assoc : \mathcal{C} \mapsto \wp(\mathbf{Name} \times N \times \mathbf{Bool} \times \mathbf{Bound})$ is a function that returns a set of association ends of a classifier. Each association end is quadruple $\langle r, pos, nv, bnd \rangle$ where r is the name of the association, pos the position of the classifier in the association, nv the navigability of the association end and bnd the multiplicity of the association end.

The partial order $\sqsubseteq$ indicates subtyping relation between classifiers in a class diagram. It is extended to include other types in an obvious manner. For instance, $integer \sqsubseteq number$ and $C \sqsubseteq Object$ for any classifier C including $Object$. Denote by $\mathbf{Name}(D)$ the set of the names of classifiers, attributes, operations and relationships specified in D. Note that $\mathbf{Name}(D)$ does not include primitive types or any classifier that is not specified in D.

Example 1. We shall use as a running example an application that calculates the total net price of a composite equipment from the net prices of its parts. Figure 1 shows the class diagram.

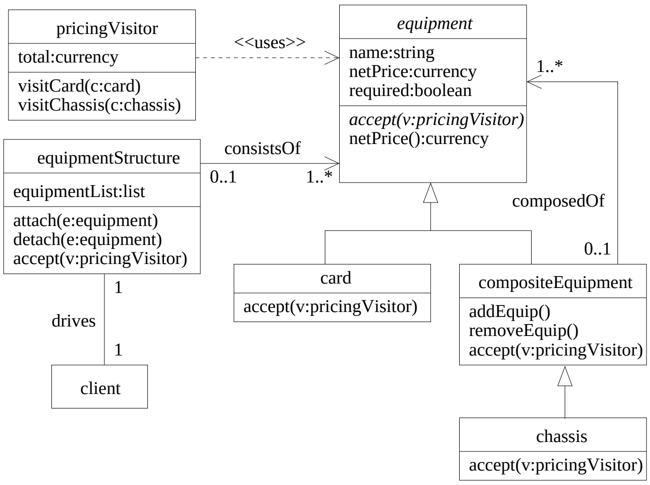

Figure 1: A Class Diagram of a Price Calculation Application

The diagram describes equipment structures that consist of cards and chassises where a chassis is a composite equipment of cards. A *PricingVisitor* object visits each element in the equipment structure and gets its net price in order to calculate the total net price of the equipment. Operations *visitCard* and *visitChassis* are used to visit *Card* and *Chassis* objects. A visited element accepts the visitor object and returns itself to the visitor. The visitor then calls the *netPrice* operation to the element to get its net price.

The formal description of the application is

$$D_a = \langle \mathcal{C}_a, \sqsubseteq_a, \mathcal{R}_a, mclass_a, attr_a, oper_a, assoc_a \rangle$$

where

$$\mathcal{C}_a = \left\{ \begin{array}{c} card, chassis, client, compositeEquipment, \\ equipment, equipmentStructure, pricingVisitor \end{array} \right\}$$

$$\sqsubseteq_a = \left\{ \begin{array}{c} \langle card, equipment\rangle, \langle compositeEquipment, equipment\rangle, \\ \langle chasis, compositeEquipment\rangle \end{array} \right\}^\star$$

$$\mathcal{R}_a = \{consistOf, composedOf, drives, uses\}$$

$$mclass_a = \left\{ \begin{array}{c} equipment \mapsto abstract, card \mapsto concrete, \\ pricingVisitor \mapsto concrete, chassis \mapsto concrete, \\ equipmentStructure \mapsto concrete, client \mapsto concrete, \\ compositeEquipment \mapsto concrete, consistOf \mapsto association, \\ composedOf \mapsto association, drives \mapsto association, \\ uses \mapsto dependency \end{array} \right\}$$

$$attr_a(equipment) = \{\langle name, String\rangle, \langle netPrice, Current\rangle, \langle required, Boolean\rangle\}$$

$$\vdots$$

$$oper_a(equipment) = \{\langle accept, PricingVisitor, void\rangle, \langle netPrice, \epsilon, Currency\rangle\}$$

$$\vdots$$

$$assoc_a(equipment) = \left\{ \begin{array}{c} \langle uses, 2, false, 1..1\rangle, \langle consistOf, 2, false, 1..many\rangle, \\ \langle composedOf, 2, false, 1..many\rangle \end{array} \right\}$$

$$\vdots$$

Values of $attr_a, oper_a$ and $assoc_a$ for other classifiers are omitted.

2.2 Structural Patterns

In this work, pattern are specified using RBML [France et al. 2004, Kim 2004] – a UML-based pattern specification language that specifies patterns as metamodels. RBML is chosen because it precisely describes pattern properties and is designed to support model-level use of design patterns as opposed to implementation level use. The roles defined in a pattern are played by model elements in a conforming model. In particular, a *Structural Pattern Specification* (SPS) in RBML captures a class diagram view of a pattern solution in terms of *classifier* and *relationship* roles whose bases are *Classifier* and *Relationship* metaclasses in the UML metamodel [Kim et al. 2003]. A classifier role is associated with a set of feature roles that determines the characteristics of the classifier role and is connected to other classifier roles by relationship roles.

A simplified SPS for the Visitor design pattern [Gamma et al. 1995] is shown in Figure 2. The Visitor pattern provides a solution for handling crosscutting

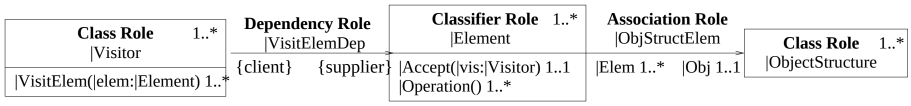

Figure 2: A Partial SPS for the *Visitor* Pattern

operations in a structure of classes called *elements* by putting these operations into separate classes called *visitors* and having the visitors visit the elements to perform the operations on the elements.

The SPS defines three classifier roles *Visitor*, *Element*, and *ObjectStructure* and two relationship roles *VisitElemDep* and *ObjStrucElem* roles. The base metaclass of role is specified above role name. For example, the *Visitor* role has the *Class* metaclass as its base, which constrains that only instances of the *Class* metaclass can play the role. The multiplicity *1..** specified at the right upper corner postulates that there must be at least one class playing the role. The *Visitor* role has a behavioral feature role *VisitElem()* with a parameter role *elem*. This further constrains instances that play the role to possess operations playing the behavioral feature role. The roles are connected by two relationship roles, the *ObjStructElem* association role and *VisitElemDep* dependency role. The *Element* and *ObjectStructure* roles can be explained similarly.

Syntactically, an SPS is a class diagram plus information about realization multiplicities of roles. The realization multiplicity of a role (shown near the role name) constrains the number of elements in a conforming class diagram that can play the role. It is different from object multiplicity of an association end which constrains the number of objects.

Definition 2. A structural pattern is a tuple $\langle D, \mu \rangle$ where D is a class diagram and $\mu : \mathbf{Name}(D) \mapsto \mathbf{Bound}$ is a function mapping a role to $\mathbf{Name}(D)$ to a multiplicity interval.

Example 2. Let ϵ denote the empty sequence. The formal description of the Visitor pattern is $\langle D_p, \mu_p \rangle$ where

$$D_p = \langle \mathcal{C}_p, \sqsubseteq_p, \mathcal{R}_p, mclass_p, attr_p, oper_p, assoc_p \rangle$$

where

$$\begin{aligned}
\mathcal{C}_p &= \{Visitor, Element, ObjectStructure\} \\
\sqsubseteq_p &= \emptyset \\
\mathcal{R}_p &= \{VisitElemDep, ObjStructElem\} \\
mclass_p &= \left\{ \begin{array}{c} Visitor \mapsto concrete, Element \mapsto classifier, \\ ObjectStructure \mapsto concrete, \\ VisitElemDep \mapsto dependency, ObjStructElem \mapsto association \end{array} \right\}
\end{aligned}$$

$$\begin{aligned}
attr_p(Element) = attr_p(Visitor) &= attr_p(ObjectStructure) = \emptyset \\
oper_p(Element) &= \{\langle Accept, Visitor, void\rangle, \langle Operation, \epsilon, void\rangle\} \\
oper_p(ObjectStructure) &= \emptyset \\
oper_p(Visitor) &= \{\langle VisitElem, Element, void\rangle\} \\
assoc_p(Element) &= \left\{ \begin{array}{c} \langle VisitElemDep, 2, false, 1..1\rangle, \\ \langle ObjStructElem, 1, true, 1..many\rangle \end{array} \right\} \\
assoc_p(ObjectStructure) &= \{\langle ObjStructElem, 2, false, 1..1\rangle\} \\
assoc_p(Visitor) &= \{\langle VisitElemDep, 1, true, 1..1\rangle\}
\end{aligned}$$

and

$$\mu_p(Visitor) = \mu_p(Element) = \mu_p(ObjectStructure) = 1..many$$

$$\mu_p(VisitElem) = \mu_p(Operation) = 1..many$$

$$\mu_p(Accept) = \mu_p(VisitElemDep) = \mu_p(ObjStructElem) = 1..many$$

2.3 Pattern Conformance

When an SPS pattern is used in an application, each role in the SPS must be played by an element in the application. The model element in the application does not usually have the same name as the role since the SPS is constructed for a wide range of applications and hence adopts names of a generic nature while the name of a model element should be more specific and meaningful to the application at hand. Our notion of conformance is parametric on a mapping ρ from the names of roles to the names of model elements, i.e., $\rho(Role) = Elem$ says that *Elem* plays the role *Role*. In addition to this name mapping, the classifier in the application may add new attributes and/or replace the type of an existing attribute by one of its subtype. Similarly, new operations can be added to a classifier and the return type of an existing operation can be made smaller, i.e., its return type can be made smaller and the type of any of its arguments can be made larger. The multiplicities of association ends can be made smaller. During reusing process, classifiers in a class diagram can be merged, split or class hierarchy can be changed. Pattern conformance requires that for each classifier c_2 in the pattern, there is a corresponding classifier c_1 in the application such that c_1 possesses each and every feature of c_2 in the original or a refined form.

The notion of conformance is based on that of a consistent role mapping from roles in a pattern to elements in an application. Subtyping in an object-oriented system allows a classifier to inherit all features of its super-classifiers. Let $D = \langle \mathcal{C}, \sqsubseteq, \mathcal{R}, mclass, attr, oper, assoc \rangle$. Then $attr^\star(c)$ is defined as the set of all attributes defined in c and all of its super-classifiers. $oper^\star(c)$ and $assoc^\star(c)$ are defined similarly.

$$attr^\star(c) = \bigcup\{attr(c') \mid c \sqsubseteq c' \wedge c' \in \mathcal{C}\}$$
$$oper^\star(c) = \bigcup\{oper(c') \mid c \sqsubseteq c' \wedge c' \in \mathcal{C}\}$$
$$assoc^\star(c) = \bigcup\{assoc(c') \mid c \sqsubseteq c' \wedge c' \in \mathcal{C}\}$$

Definition 3. Let $D_a = \langle \mathcal{C}_1, \sqsubseteq_1, \mathcal{R}_1, mclass_1, attr_1, oper_1, assoc_1 \rangle$ and $D_p = \langle \mathcal{C}_2, \sqsubseteq_2, \mathcal{R}_2, mclass_2, attr_2, oper_2, assoc_2 \rangle$ be class diagrams. A function $\rho : \mathbf{Name}(D_p) \mapsto \mathbf{Name}(D_a)$ is a consistent role mapping from D_p and D_a provided that the following conditions both hold.

1. If $(c_1 \in \mathcal{C}_1) \wedge (c_2 \in \mathcal{C}_2) \wedge (c_1 = \rho(c_2))$ then
   (a) $mclass_1(c_1) \preceq mclass_2(c_2)$;
   (b) for each attribute $\langle a_2, T_2 \rangle \in attr_2^\star(c_2)$, there is an attribute $\langle a_1, T_1 \rangle \in attr_1^\star(c_1)$ such that $a_1 = \rho(a_2)$ and $T_1 \sqsubseteq_1 \rho(T_2)$; [3]
   (c) for each operation $\langle o_2, Ta_2, Tr_2 \rangle \in oper_2^\star(c_2)$, there is an operation $\langle o_1, Ta_1, Tr_1 \rangle \in oper_1^\star(c_1)$ such that $o_1 = \rho(o_2)$, and $Tr_1 \sqsubseteq_1 \rho(Tr_2)$ and there is a sub-sequence Ta'_1 of Ta_1 such that $\rho(Ta_2(i)) \sqsubseteq_1 Ta'_1(i)$ for all $1 \leq i \leq |Ta_2|$; and
   (d) for each association end $\langle r_2, pos_2, nv_2, bnd_2 \rangle \in assoc_2^\star(c_2)$, there is an association end $\langle r_1, pos_1, nv_1, bnd_1 \rangle \in assoc_1^\star(c_1)$ such that $r_1 = \rho(r_2)$, $pos_2 = pos_1$, $nv_2 = nv_1$ and $bnd_1 \subseteq bnd_2$ where $bnd_1 \subseteq bnd_2$ holds whenever bnd_1 is contained in bnd_2; and
2. If $(r_1 \in \mathcal{R}_1) \wedge (r_2 \in \mathcal{R}_2) \wedge (r_1 = \rho(r_2))$ then
   (a) $mclass_1(r_1) \preceq mclass_2(r_2)$; and
   (b) for each classifier $c_2 \in \mathcal{C}_2$ such that $\langle r_2, pos, \_, \_ \rangle \in assoc_2^\star(c_2)$ for some $pos \in N$, there is a classifier $c_1 \in \mathcal{C}_1$ such that $\langle r_1, pos, \_, \_ \rangle \in assoc_1^\star(c_1)$ and $c_1 = \rho(c_2)$ where _ stand for a don't care value.

[3] The function ρ is extended by $\rho(T) = T$ for any $T \notin \mathbf{Name}(D_p)$.

Some explanations are in order. A consistent role mapping ρ tells which elements in D_a play which roles in D_p. It ensures that each role in D_p is played by an element in D_a. It allows multiple model elements to play a single role but does not allow a single model element to play multiple roles. This issue is resolved by considering all consistent role mappings. The condition (1) says that a classifier c_1 in D_a plays a classifier role c_2 in D_p only if each feature of c_2 is played by a feature of c_1. The condition (2) states that an association r_1 in D_a plays an association role r_2 in D_p only if each participant of r_2 is played by its corresponding participant of r_1.

Example 3. The following is a consistent role mapping from D_p in Example 2 to D_a in Example 1.

$$\left\{\begin{array}{l} \mathit{Accept} \mapsto \mathit{accept}, Visitor \mapsto pricingVisitor, VisitElemDep \mapsto \mathit{uses}, \\ \mathit{ObjectStructure} \mapsto \mathit{equipmentStructure}, ObjStructElem \mapsto consistsOf, \\ \mathit{Element} \mapsto \mathit{chassis}, VisitElement \mapsto visitChassis, \\ \mathit{Operation} \mapsto addEquip \end{array}\right\}$$

Given a class diagram D_a and a structural pattern $\langle D_p, \mu\rangle$, the set of all consistent role mappings between D_a and D_p is finite since $\mathbf{Name}(D_a)$ and $\mathbf{Name}(D_p)$ are both finite.

Definition 4. Let $D_a = \langle \mathcal{C}_1, \sqsubseteq_1, \mathcal{R}_1, mclass_1, attr_1, oper_1, assoc_1\rangle$ and $D_p = \langle \mathcal{C}_2, \sqsubseteq_2, \mathcal{R}_2, mclass_2, attr_2, oper_2, assoc_2\rangle$ be class diagrams. Let $\rhd \subseteq \mathbf{Name}(D_a) \times \mathbf{Name}(D_p)$ be defined by $e \rhd r$ iff $e = \rho(r)$ for some consistent role mapping ρ from D_p and D_a. Then D_a is said to conform to $\langle D_p, \mu\rangle$ if

1. $\|\{c_1 \mid c_1 \rhd c_2\}\| \in \mu(c_2)$ for each $c_2 \in \mathcal{C}_2$ where $\|S\|$ is the cardinality of a set S.
2. If $c_1 \rhd c_2$ and $\langle a_2, \_\rangle \in attr_2^\star(c_2)$ then $\|\{a_1 \mid a_1 \rhd a_2 \wedge \langle a_1, \_\rangle \in attr_1^\star(c_1)\}\| \in \mu(a_2)$.
3. If $c_1 \rhd c_2$ and $\langle o_2, \_, \_\rangle \in oper_2^\star(c_2)$ then $\|\{o_1 \mid o_1 \rhd o_2 \wedge \langle o_1, \_, \_\rangle \in oper_1^\star(c_1)\}\| \in \mu(o_2)$.
4. If $c_1 \rhd c_2$ and $\langle r_2, pos, \_, \_\rangle \in assoc_2^\star(c_2)$ then $\|\{r_1 \mid r_1 \rhd r_2 \wedge \langle r_1, pos, \_, \_\rangle \in assoc_1^\star(c_1)\}\| \in \mu(r_2)$.

The condition (1) says that the number of classifiers in D_a playing a classifier role c_2 in D_p must be within the interval $\mu(c_2)$. The conditions 2-4 states the similar requirement on the number of features in a classifier in D_a that plays a classifier in D_p.

The relation $\rhd$ in the above definition can be calculated by computing all consistent role mappings between D_a and D_p and combining them as sets of bindings via set union. Once $\rhd$ is computed, verifying conditions 1-4 is simple. Thus, verification of the conformance of D_a to $\langle D_p, \mu\rangle$ boils down to computing consistent role mappings from D_p to D_a which is the subject of the next section.

3 Computing Consistent Role Mappings

A prerequisite for any automated tool for reasoning about UML design models is a representation scheme for model elements. A logic program is declarative in that it describes what a problem is but not how it can be solved. Solutions to the problem can be generated by a logic programming language system such as Prolog. The domain of the problem is described as a collection of logic statements and so is the problem. Logic inference capability of a logic programming language such as Prolog can not only check if the problem is solvable, but also can find all solutions to the problem. This capability is used to compute consistent role mappings.

3.1 Class Diagrams as Logic Programs

This section presents a scheme for representing model elements of a UML class diagram and its associated OCL constraints as Prolog statements.

3.1.1 Metamodel knowledge

The partial order $\langle \mathbf{MetaClass}, \preceq\rangle$ represents metamodel knowledge which is common to a range of applications. They are represented as follows. Each element $k \in \mathbf{MetaClass}$ corresponds to a predicate named k and $\langle \mathbf{MetaClass}, \preceq\rangle$ is represented as a set of Prolog rules of the form $k_2(X)$:-$k_1(X)$ for each pair consisting of k_1 and k_2 such that $k_1 \preceq k_2$ and there is not any k' such that $k_1 \preceq k' \preceq k_2$. The example partial order $\langle \mathbf{MetaClass}, \preceq\rangle$ in Section 2 is represented by the following Prolog rules.

```
classifier(X) :- interface(X).      classifier(X) :- class(X).
class(X) :- abstract_class(X).      class(X) :- concrete_class(X).
association(X) :- dependency(X).    dependency(X) :- usage(X).
```

The knowledge that the sub-typing relation is transitive is encoded by this Prolog rule: *is_a(X,Y) :- is_a(X,Z), is_a(Z,Y)*. Since inheritance is non-cyclic, a call to *is_a* with its first argument being a ground term is guaranteed to terminate universally. The translation from patterns to queries ensures that *is_a* is always called with its first argument ground. Sub-typing induces a rule for inherited features:

```
has_feature(T,F) :- is_a(T,T1),has_feature(T1,F).
```

The following rules realize the containment relation between pairs of object multiplicity bounds.

```
bound_subset(bounds(L1,U1),bounds(L2,U2)) :-
    bound_leq(L2,L1), bound_leq(U1,U2).

bound_leq(B1,B2) :-
    B2 == many -> true; B1 \== many, B1 =< B2.
```

The predicate *bound_subset* will only be used to check if a given pair of bounds is contained in another given pair of bounds; it will not be used to generate the containment relation between pairs of bounds.

The Prolog rules representing meta-model knowledge are completed with Prolog facts that are specific to a model.

3.1.2 Sub-typing

A sub-typing relation $\sqsubseteq$ is represented by a predicate *is_a* such that *is_a*(T1,T2) indicates that T1 is a sub-type of T2. The sub-typing relation $\sqsubseteq_a$ in Example 1 is represented as follows.

```
is_a(compositeEquipment,equipment).
is_a(chassis,compositeEquipment).
is_a(card,equipment).
```

3.1.3 Metaclassing

A meta-classing function *mclass* is translated to a set of Prolog facts. Each binding of the form $e \mapsto k$ in *mclass* is translated into a fact $k(e)$. For instance, the meta-typing $mclass_a$ in Example 1 is represented by the following Prolog facts.

```
abstract(equipment).                association(consistOf).
concrete(card).                     association(drives).
concrete(pricingVisitor).           association(composedOf).
concrete(chassis).                  dependency(uses).
concrete(equipmentStructure).
concrete(compositeEquipment).
concrete(client).
```

3.1.4 Features of type

A classifier is defined in terms of attributes, operations and association ends. These are called features of the classifier. That a classifier has a feature is represented as a fact of the form *has_feature(CName, Info)* where *CName* is the name of the classifier and *Info* is a ground term describing the feature.

3.1.5 Operations and Attributes

That a classifier has an operation is represented as a Prolog fact *has_feature(CName, op(OpName, ArgTypes, RType)). CName* is the name of the classifier, *OpName* the name of the operation, *ArgTypes* the list of its argument types and *RType* the type of its returned value. For example, the only operation defined in the *chassis* class in Figure 1 is represented by

```
has_feature(chassis,op(accept,[pricingVisitor],void))
```

and the operations defined in the *equipment* class are encoded as these Prolog facts.

```
has_feature(equipment,op(accept,[pricingVisitor],void)).
has_feature(equipment,op(netPrice,[],void)).
```

Attributes are treated similarly. That a classifier has an attribute is represented as a Prolog fact of the form *has_feature(CName,attr(AttrName,AttrType))* where *CName* is the name of the classifier, *AttrName* the name of the attribute and *AttrType* the type of the attribute. For instance, the attributes of the *equipment* class are encoded as these Prolog facts.

```
has_feature(equipment,attr(name,string)).
has_feature(equipment,attr(netPrice,currency)).
has_feature(equipment,attr(required,boolean)).
```

3.1.6 Association Ends

An association is defined in terms of association ends that may be annotated with object multiplicity and navigability constraints. An association is uniquely identified with its ends. Thus, an association is represented by representing its ends. In addition to its meta-class information, an association end of a classifier is represented by a fact *has_feature(CName, assoc(AssocName, Pos, Navigability, bounds(Lower,Upper)))* where *CName* is the name of the classifier that participates in the association at the end. *Pos* is the position of the classifier in the association relationship. *AssocName* is the name of the association. *Navigability* is either *true* or *false*, indicating whether the end is navigable. *Lower* is the lower bound on the object multiplicity at the end and *Upper* the upper bound. For instance, the association *composedOf* is represented by these two facts:

```
has_feature(equipment,
    assoc(composedOf,2,false,bounds(1,many))).
has_feature(compositeEquipment,
    assoc(composedOf,1,true,bounds(0,1))).
```

The only dependency *uses* in Figure 1 is represented by these two facts.

```
has_feature(equipment,assoc(uses,1,true,bounds(1,1))).
has_feature(pricingVisitor,assoc(uses,2,false,bounds(1,1))).
```

3.2 Patterns as Queries

Our goal is to discover a mapping from pattern roles to model elements such that when the roles are substituted by the model elements, the pattern is satisfied by the model. For this purpose, we represent a design pattern as a query. The representation uses the same predicates for representing UML models. Each role is represented as a variable. Roles except association end roles are represented as atoms in the same way as their corresponding model elements are represented as facts. For instance, the three classifier roles in Figure 2 are represented as *class(Visitor), class(ObjectStructure)* and *classifier(Element)* respectively. That *Visitor* has an unary behavioral role *VisitElem* with an argument of type *Element* is represented as *has_feature(Visitor,op(VisitElement,[Element],void))*.

Each association end role is represented by two atoms. For instance, the association end role *Obj* is represented by these two atoms

```
has_feature(ObjectStructure,assoc(ObjStructElem,2,false,ObjBnds)),
bound_subset(ObjBnds,bounds(1,1))
```

where *ObjBnds* is a variable not appearing elsewhere. The role *Obj* does not appear in this representation since it is uniquely determined by the roles *ObjectStructure* and *ObjStructElem*. In fact, an association end needs not be named. Observe that *ObjBnds* is the pair of object multiplicity bounds for the model element that plays the *Obj* role and that *bound_subset(ObjBnds,bounds(1,1))* checks if *ObjBnds* is contained in the pair of object multiplicity bounds for the *Obj* role.

Each pattern role is represented as one or two atoms. The conjunction of the atoms obtained from all pattern roles forms a query. The query representing the example pattern in Fig. 2 is

```
   % sub-typing relation is empty
   % meta-class information
classifier(Element),
concrete_class(Visitor),
concrete_class(ObjectStructure),
dependency(VisitElemDep),
association(ObjStructElem),
   % features of Element
has_feature(Element,op(Accept,[Visitor],void)),
has_feature(Element,op(Operation,[],void)),
```

```
has_feature(Element,assoc(VisitElemDep,1,true,SuplierBnds)),
bound_subset(SupplierBnds,bounds(1,1)),
   % features of Visitor
has_feature(Visitor,op(VisitElement,[Element],void)),
has_feature(Visitor,assoc(VisitElemDep,2,false,ClientBnds)),
bound_subset(ClientBnds,bounds(1,1)),
has_feature(Element, assoc(ObjStructElem,1,true,ElemBnds)),
bound_subset(ElemBnds,bounds(1,many)),
    % features of ObjectStructure
has_feature(ObjectStructure,assoc(ObjStructElem,2,false,ObjBnds)),
bound_subset(ObjBnds,bounds(1,1)).
```

3.3 Inference of Consistent Role Mappings

The logic program represents elements in a UML model and their relationships, whilst the query represents roles in a pattern and their relationships. This facilitates the calculation of consistent role mappings because they can be found by executing the program and query. The following theorem states that the set of all consistent role mappings can be obtained by computing all computed answers to the query with the program and projecting the computed answers to the set of the variables that represent roles. In addition, the LD-resolution of the query with the program will always terminate.

Theorem 1 *Let P denotes the program, Q the query and V the set of variables representing roles. Then*

(a) The LD-resolution of $P \cup \{\leftarrow Q\}$ universally terminates.

(b) A substitution θ is a computed answer to $P \cup \{\leftarrow Q\}$ via LD-resolution if and only if $\theta \uparrow V$ is a consistent mapping where $\theta \uparrow V$ is θ restricted to V.

Proof. *Consider (a) first. All calls except those to* bound_subset*/2 universally terminates. Calls to* interface*/1,* class*/1 and* classifier*/1 obviously terminate universally since these predicates are not recursive. Calls to* is_a*/2 universally terminates since sub-typing relations is not cyclic. This together with the fact that all the rules in the program representing the model do not have any function symbol, implies that calls to* has_feature*/2 universally terminates. Note that all facts in the program representing the model are ground, and all variables in the head of a rule also appear in the body of the rule. Therefore, the successful execution of a call to* interface*/1,* class*/1,* classifier*/1 and* has_feature*/2 grounds all its arguments. By construction, for each call* bound_subset*(Bs1,Bs2) in Q, Bs2 is a ground term and Bs1 occurs in a call that precedes* bound_subset*(Bs1,Bs2), thus, both Bs1 and Bs2 are ground upon the selection of* bound_subset*(Bs1,Bs2)* *by the LD-resolution, which implies that all computed answers to $P \cup \{\leftarrow Q\}$ are ground substitutions. The set of variables Q consists of those variables representing roles and those variables representing pairs of bounds. Then the proof of (b) follows directly from the soundness and the completeness of the LD-resolution procedure [Lloyd 1987].*

We have constructed a conformance verifier in Prolog that first computes all consistent role mappings from the class diagram in the structural pattern to the class diagram in the model and then checks if the conditions in Definition 4 hold.

Example 4. Let P be the program that represents the class diagram D_a in Example 1 and Q the query that represents the pattern D_p in Example 2. Let $\mathcal{X} \uplus \mathcal{Y} = \{X \cup Y \mid X \in \mathcal{X} \wedge Y \in \mathcal{Y}\}$. Our conformance verifier first computes the following set of consistent role mappings by executing P with the query Q.

$$
\begin{array}{l}
\{\{Accept \mapsto accept, Visitor \mapsto pricingVisitor, VisitElemDep \mapsto uses\}\} \uplus \\
\left\{\begin{array}{l}
\left\{\begin{array}{l} ObjectStructure \mapsto equipmentStructure, \\ ObjStructElem \mapsto consistsOf \end{array}\right\}, \\
\left\{\begin{array}{l} ObjectStructure \mapsto compositeEquipment, \\ ObjStructElem \mapsto composedOf \end{array}\right\}, \\
\{ObjectStructure \mapsto chassis, ObjStructElem \mapsto composedOf\}
\end{array}\right\} \uplus \\
\left\{\begin{array}{l}
\left\{\begin{array}{l} Element \mapsto chassis, VisitElement \mapsto visitChassis, \\ Operation \mapsto addEquip \end{array}\right\}, \\
\left\{\begin{array}{l} Element \mapsto chassis, VisitElement \mapsto visitChassis, \\ Operation \mapsto removeEquip \end{array}\right\}, \\
\left\{\begin{array}{l} Element \mapsto chassis, VisitElement \mapsto visitChassis, \\ Operation \mapsto netPrice \end{array}\right\}, \\
\{Element \mapsto card, VisitElement \mapsto visitCard, Operation \mapsto netPrice\}
\end{array}\right\}
\end{array}
$$

The conformance verifier then checks conditions 1-4 in Definition 4 and finds that they all hold. Therefore, the class diagram conforms to the pattern.

4 Related Work

There has been much work on detecting pattern instances in code. Albin-Amiot and Guéhéneuc [Albin-Amiot et al. 2001] propose a meta-modeling approach to define and detect design patterns in Java code by structural matching. Balanyi and Ference [Balanyi et al. 2003] use a XML-based language to represent design patterns and detect pattern instances in C++ code. Fabry and Mens [Fabry & Mens 2004] use logic meta programming to detect design patterns in different languages (e.g., Java, Smalltalk). Heuzeroth *et al.* define design patterns in a tuple of classes, methods, and attributes and use them to find pattern instances in Java code using pattern-specific algorithms. These works support the reverse engineering efforts at the programming level for understanding legacy systems and improving their quality attributes.

Kramer and Prechelt [Kraemer and Prechelt 1996] and Bergenti and Poggi [Bergenti and Poggi 2000] propose Prolog-based approaches where design patterns are represented as Prolog rules which are used to search pattern instances in Prolog programs translated from application models. A limitation of these approaches is that the pattern solution structures used to build the Prolog rules are, in fact, a typical instance of the patterns, and it is rare to find such models that have the same instance, which significantly limits their applicability.

Metric-based approaches are studied in [Antoniol et al. 1998], [Shull et al. 1996]. Antoniol *et. al.* [Antoniol et al. 1998] use simple class-level metrics (e.g., the number of attributes, the number of operations, the number of different types of relationships) as constraints to recover design patterns in designs. Their approach is similar to the approaches based on minimal key structures in that the structural constraints are expressed in metrics. More sophisticated metrics are used in Shull *et al.*'s work [Shull et al. 1996]. They define a design pattern by metrics in three categories of object-oriented metrics, structural metrics, and procedural metrics and use them in six steps of a searching algorithm. There is no tool support for their approach, and it is hard to see how the algorithm can be automated.

Similar to our approach, Guennec *et al.* [Guennec et al. 2000] use roles (e.g., classifier roles, feature roles) to define patterns where roles are played by UML model elements. In their work, a UML model is said to conform to a pattern if the names of model elements match to the role names. Using their notion of pattern conformance, it is hard to expect that the elements in a model have the same name as the role name unless the designer is assumed to have knowledge of their pattern specifications and intentionally uses the role names, which is not a valid assumption. Their pattern specifications also have other important properties such as role types and behaviors, but these properties are not considered in the notion of pattern conformance. Our technique establishes a precise notion of pattern conformance and enables rigorous evaluation of pattern conformance without requiring designers to have knowledge of the pattern.

Potential of logic programming as a reasoning tool in software engineering has been recognized before [Abreu 2000, Alghathbar et al. 2005, Mens 2002, Wang et al. 2004, Wuyts 1998, Zisman and Kozlenkov 2003]. To the best of our knowledge, none of previous works address the issue of conformance of a UML model to a given design pattern. Abreu reports a university information system that describes classes, inheritance, attributes and the values used to populate the classes as description logic formulae [Abreu 2000]. The description logic formulae are used to generate more efficient and specific representations for actual use. The emphasis of the work in [Abreu 2000] is to substitute description logic formulae for UML models. Our work focuses on formal reasoning about UML models, in particular, conformance of UML models to design patterns.

Wang *et. al* use constraint logic programming for symbolic execution of requirements described as live sequence charts [Wang et al. 2004]. Data variables in live sequence charts are represented as logical variables while control variables in live sequence charts as constraints. A truly symbolic execution of live sequence charts is realized by making use of two basic capabilities of a constraint logic programming language: unification and constraint propagation. The work in [Wang et al. 2004] allows software designer to play with his design whilst our work verifies if his design conforms to a given design pattern and informs him how it conforms to the design pattern.

Zisman and Kozlenkov represent elements in an UML metamodel as axioms and those in an UML model as facts [Zisman and Kozlenkov 2003]. They use a knowledge base engine based on abduction to discover and analyze structural and behavioral inconsistencies within or between UML specifications. FlowUML [Alghathbar et al. 2005] uses Horn clauses to specify information flow polices that can be checked against flow information extracted from UML sequence diagrams. These works are mainly concerned with checking consistency within and between UML models. Our work goes beyond that by inferring how a UML model conforms to a given design pattern.

Wuyts proposed a logic meta-programming language SOUL for representing structural relationships in class-based object-oriented systems [Wuyts 1998]. A declarative framework based on SOUL was constructed to reason about the structure of Smalltalk programs. SOUL was also used by Mens et. al [Mens 2002] to manage intentional source code views. A careful study of the representation proposed in [Wuyts 1998] reveals that it does not permit inference of design pattern instances in a UML model. For instance, that a class named c has a method named m is represented as a fact *method(c,m).* Without information about the types of the arguments and the returned value of the method, precise matching between a method role and a method is not possible.

5 Conclusion

We have formalized class diagrams and structural patterns as mathematical objects and given a precise definition of a notion of structural pattern conformance. We have also presented a rigorous technique for evaluating structural conformance of UML class diagrams to a structural pattern specification using logic programming. We have demonstrated how the technique can be used through the *Visitor* pattern and a model of a price calculation application. The technique can be also used to find instances of domain-specific patterns in a particular domain (e.g., telecommunication, security). We are currently applying the technique to verify valid instances of access control patterns (e.g., RBAC, MAC, DAC) for designs of access control systems in the security domain. The technique can be also used in the area of pattern-based model refactoring [France et al. 2003] for finding applicable design patterns for a given problem model. If the problem model conforms to the problem specification of a pattern, the solution of the pattern can be applied to the model.

In the subsequent work, we plan to develop tool support for the technique to translate a pattern specification to a query and a UML model into a logic program. We also plan to extend the technique to include checking semantic conformance of behavioral properties. Examples of such properties are pre- and post-conditions in behavioral features roles and the interactions among pattern elements specified in Interaction Pattern Specifications in RBML.

References

[Abreu 2000] Abreu, S.: "A Logic-based Information System"; Proc. 2^{nd} Int. Workshop on Practical Aspects of Declarative Languages, (2000), 141-153.

[Albin-Amiot et al. 2001] Albin-Amiot, H., Gueheneuc Y. G.: "Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis"; Proc. 1^{st} ECOOP Workshop on Automating Object-Oriented Software Development Methods, (2001).

[Alghathbar et al. 2005] Alghathbar, K., Wijesekera, D., Farkas, C.: "Secure UML Information Flow using FlowUML"; Proc. 3^{rd} Int. Workshop on Security in Information Systems, (2005), 229-238.

[Antoniol et al. 1998] Antoniol, G., Fiutem, R., Cristoforetti, L.: "Design Pattern Recovery in Object-Oriented Software", Proc. 6^{th} Int. Workshop on Program Comprehension, (1998), 153-160.

[Balanyi et al. 2003] Balanyi, Z., Ferenc, R.: "Mining Design Patterns from C++ Source Code"; Proc. ICSM, (2003), 305-314.

[Bergenti and Poggi 2000] Bergenti, F., Poggi, A.: "Improving UML Design Using Automatic Design Pattern Detection"; Proc. 12^{th} SEKE, (2000), 336-343.

[Brown 1996] Brown, K.: "Design Reverse-Engineering and Automated Design Pattern Detection in SmallTalk"; Dept. of Comp. Eng., North Carolina State U., (1996).

[Cunningham and Beck 1986] Cunningham, W., Beck, K: "A Diagram for Object-Oriented Programs"; ACM Sigplan Notices, 21, 11, (1986), 361-367.

[Eden 1999] Eden, A.: "Precise Specification of Design Patterns and Tool Support in Their Application"; U. of Tel Aviv, Israel, (1999).

[Fabry & Mens 2004] Fabry, J., Mens, T.: "Language-Independent Detection of Object-Oriented Design Patterns"; Computer Languages, Systems & Structures, 30, 1-2, (2004), 21-33.

[France et al. 2003] France, R., Ghosh, S., Song, E., Kim, D.: "A Metamodeling Approach to Pattern-Based Model Refactoring"; IEEE Soft. Special Issue on Model Driven Development, 20, 5, (2003), 52-58.

[France et al. 2004] France, R., Kim, D., Ghosh, S., Song, E.: "A UML-Based Pattern Specification Technique"; IEEE Tran. Soft. Eng. 30, 3, (March, 2004), 193-206.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software"; Addison-Wesley, (1995).

[Guennec et al. 2000] Guennec, A. L., Sunye, G., Jezequel, J.: "Precise Modeling of Design Patterns"; Proc. UML, (2000), 482-496.

[Heuzeroth et al. 2003] Heuzeroth, D., Holl, T., Högström, G., Löwe, W.: "Automatic Design Pattern Detection"; Proc. 11^{th} IWPC, (2003), 94-103.

[Keller et al. 1999] Keller, R. K., Schauer, R. Robitaille, S., Page, P: "Pattern-Based Reverse Engineering of Design Components"; Proc. 21^{st} ICSE, (1999), 226-235.

[Kim 2004] Kim, D.: "A Meta-Modeling Approach to Specifying Patterns"; Colorado State U., Fort Collins, CO, (2004).

[Kim et al. 2003] Kim, D., France, R., Ghosh, S., Song, E.: "A Role-Based Metamodeling Approach to Specifying Design Patterns"; Proc. 27^{th} COMPSAC, (2003), 452-457.

[Kim and Lu 2006] Kim, D., Lu, L. : "Inference of Design Pattern Instances in UML models via Logic Programming"; Proc. 11^{th} ICECCS, (2006), 47-56.

[Kraemer and Prechelt 1996] Kraemer, C., Prechelt, L.: "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software"; Proc. 3^{rd} Working Conference on Reverse Engineering, (1996), 208-215.

[Lano et al. 1996] Lano, K., Bicarregui, J., Goldsack, S.: "Formalising Design Patterns"; Proc. 1^{st} BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Comp. Sci., Springer, (1996).

[Lauder and Kent 1998] Lauder, A., Kent, S.: "Precise Visual Specification of Design Patterns"; Proc. ECOOP, (1998), 114-136.

[Lloyd 1987] Lloyd, J. W.: "Foundations of Logic Programming"; Springer-Verlag, (1987).

[Lu 2005] Lu, L.: "Use of Assertions in Declarative Diagnosis"; Proc. 20^{th} ACM Symp. on Applied Computing, (2005), 1404-1408.

[Mai and Champlain 2001] Mai, Y., de Champlain, M.: "A Pattern Language to Visitors"; Proc. 8^{th} PLoP, (2001).

[Mapelsden et al. 2002] Mapelsden, D., Hosking, J., Grundy, J.: "Design Pattern Modelling and Instantiation using DPML"; Proc. 40^{th} TOOLS, ACS, (2002), 3-11.

[Mens 2002] Mens, K., Mens, T., Wermelinger, M.: "Maintaining software through intentional source-code views"; Proc. 14^{th} SEKE, (2002), 289-296.

[Mikkonen 1998] Mikkonen, T.: "Formalizing Design Patterns"; Proc. the 20^{th} ICSE, (1998), 115-124.

[OMG 2003] , The Object Management Group (OMG): "Unified Modeling Language"; OMG, Version 1.5, (March, 2003), `http://www.omg.org`.

[Philippow et al. 2003] Philippow, I., Streitferdt, D., Riebisch, M.: "Design Pattern Recovery in Architectures for SupportingProduct Line Development and Application"; Modelling Variability for Object-Oriented Product Lines, BookOnDemand Publ. Co, (2003), 42-57.

[Prechelt et al. 2002] Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W. F.: "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance"; IEEE Trans. on Soft. Eng., 28,6, (2002), 595-606.

[Shapiro 1982] Shapiro, E.: "Algorithmic Program Diagnosis"; Proc. 9^{th} POPL (1982), 299-308.

[Shull et al. 1996] Shull, F., Melo, W. L., Basili, V. R.: "An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems", Tech. Rep. UMIACS-TR-96-10, U. of Maryland, (1996).

[Wang et al. 2004] Wang, T., Roychoudhury, A., Yap, R.H.C., Choudhary, S. C.: "Symbolic Execution of Behavioral Requirements"; Proc. 6^{th} Int. Symp. on Practical Aspects of Declarative Languages, (2004), 178-192.

[Warmer and Kleppe 2003] Warmer, J., Kleppe, A.: "The Object Constraint Language Second Edition: Getting Your Models Ready for MDA"; Addison Wesley, (2003).

[Wuyts 1998] Wuyts, R.: "Declarative Reasoning about the Structure Object-Oriented Systems"; Proc. TOOLS, (1998), 112-124.

[Zisman and Kozlenkov 2003] Zisman, A., Kozlenkov, A.: "managing Inconsistencies in UML Specifications"; Proc. 4^{th} Int. Conf. on Soft. Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing, ACIS, (2003), 128-138.