9

USING ROLES TO CHARACTERIZE MODEL FAMILIES

Robert B. France, Dae-Kyoo Kim
Eunjee Song, Sudipto Ghosh
Colorado State University, Fort Collins, CO 80523, USA
{*france, dkkim, song, ghosh*}*@cs.colostate.edu*

ABSTRACT

The development of reusable requirements and design artifacts often requires one to characterize families of problem and solution models. This paper presents a metamodeling approach to characterizing a family of models. A characterization is expressed as a Role Model that consists of roles that can be played by UML model elemets. In this paper we describe how a family of UML static structural diagrams that have the structural properties defined by a pattern can be characterized by a Static Role Model (SRM). The Abstract Factory pattern is used to illustrate how SRMs can be used to specify reusable designs expressed as patterns.

1 INTRODUCTION

To support the use of patterns in model refactoring, patterns should be capable of describing solutions in modeling terms. Pattern-based model refactoring occurs when a pattern is incorporated into a model. The use of the UML in industry is growing, thus we are interested in developing pattern specifications that support pattern-based model refactoring of UML models.

A pattern specification is defined in this paper as a characterization of a family of UML design models that have behavioral and structural properties specified by a pattern. A design model is said to *realize* (or *conform* to) a

pattern specification if it satisfies the properties defined by the pattern specification. Pattern specifications can be viewed as metamodels in the sense that they determine a family of models (the pattern realizations). A realization can have properties not specified in a pattern specification (i.e., additional properties that do not contradict pattern properties), and the properties stated in a pattern specification can be realized in different ways across conforming models. The following concerns guide our work on developing a precise notation for expressing pattern specifications:

- In order to leverage UML tools, the pattern specification notation should be based on the UML infrastructure where possible. The concepts used in our work relate to concepts defined at the UML metamodel level (level M2) [OMG2001]. We also use UML graphical notation to represent role structures, thus UML tools can be used to build the graphical forms of our pattern specifications.

- The pattern specification approach should support a well-defined notion of pattern conformance. A modeler should be able to determine that a model conforms to a pattern specification. Furthermore, it should be possible to automate non-trivial aspects of the conformance checking procedure.

In this paper, we present an approach to expressing pattern specifications as characterizations of model families. The metamodeling approach presented in this paper allows one to create metamodels, called *Static Role Models* (SRMs), that characterize UML static structural models, that is, models that depict classifiers (e.g., UML class, interface, type constructs) and their relationships with each other (e.g., using UML association, dependency, generalization constructs) [OMG2001]. The approach is based on the notion of a (model element) *role*, where a role is a specification of properties possessed by a family of UML model elements.

In section 2 we give an overview of other works on precisely defining patterns and relate them to our approach. In section 3 we describe how SRMs can be used to characterize families of static structural models and illustrate the approach using the Abstract Factory design pattern [GHJV1995] in section 4. In section 5 we introduce the notion of specializations of SRMs. We conclude in section 6 with an overview of our plans to further evolve this work.

2 RELATED WORK

Lauder and Kent [LK1998] propose an approach to presenting patterns precisely and visually using graphical constraint diagrams. In their work, patterns are described in terms of three layers of models: *role-model*, *type-model* and *class-model*. A role-model describes the essential aspects of a pattern in terms

of highly abstract state and behavior elements. A type-model is a refinement of a role-model in that it refines the role-model state and behavior elements in terms of types that abstractly specify domain realizations of the role-model. A class-model is a deployment of a type-model in terms of concrete classes. In their work, pattern realization is viewed as a refinement process in which a high-level pattern description is refined to a model realization. Establishing that a model conforms to a pattern (as expressed by a role-model) involves establishing refinement relationships across the model levels. Providing non-trivial automated support for such conformance checking requires support for proof generation/checking. Furthermore, the authors use a graphical form of constraints that is appealing but is not currently integrated with the UML and it is not clear how tools can support the notation.

Guennec et al. [GSJ2000] use a metamodeling approach in which pattern properties are expressed in terms of *meta-collaborations* that consist of collaboration roles that are played by UML modeling constructs (i.e., instances of classes in the UML metamodel at level M2 of the UML 4-layer language infrastructure). They accurately point out deficiencies in the UML notion of role models and provide an alternative representation in terms of meta-collaborations that utilize a family of recurring properties initially proposed by Eden in [Eden1999]. Unfortunately, their paper does not describe how properties (other than hierarchical structures of classifiers) are specified in their approach, nor is there a clear notion of what it means for a model to satisfy a role model.

In our work Role Models characterize families of models. Like the Lauder and Kent work, type models can realize our Role Models and they can be refined to produce more concrete realizations. It should be noted that one can also create realizations of our Role Models that are more concrete than type models. Unlike the Lauder and Kent work, our Role Model constraints are expressed in terms of the UML metamodel.

Like the Guennec et al. [GSJ2000] work, our approach is based on the UML metamodel. Unlike their work, we discuss the form and nature of constraints that can be associated with these models.

Our notion of role differs from object-based notions of role defined in RM-ODP [ISO/IEC1996, ISO/IEC2001] and by Dirk Riehle and others (e.g. see [Riehle1998]). A role in RM-ODP is an identifier of a behavior which is associated with an object. A role in Dirk's work is an entity that is realized (or is played) by an object. In our work, a role is an entity that is realized by a syntactic model construct (e.g., a class construct).

The RM-ODP notion of a template can be viewed as a specialized form of our Role Model. In RM-ODP, a template is a parametric specification of features that are common to a collection of entities. Role Models can be specialized to the point that variations can be encapsulated in parameters. These specialized Role Models can be viewed as template forms of UML models.

3 ROLE MODELS

In this work, a pattern specification consists of *Role Models*, where a Role Model is a characterization of a family of UML design models. A Role Model determines a specialization of the UML metamodel, that is, it determines a sub-language of the UML. All models that can be expressed in the sub-language are considered to be realizations of the pattern (see Section 3.3).

A Role Model specifies a structure of UML model elements in terms of a structure of (model element) roles. A role specifies a set of UML model elements of a particular type. A role type is referred to as the role's *base*, and is a class in the UML metamodel (e.g., *Class*, *Generalization*). The properties defined in a role determine a subset of the instances of the role's base. For example, a role with the *Class* base determines a subset of UML class constructs. An instance of a role's base that has the properties specified in a role can *play the role*, that is, it is a *realization* of the role. A Role Model *realization* is a model (e.g., a static structural diagram, sequence diagram) that consists of realizations of the roles in the Role Model.

Role : A role is a specification of a subset of UML model elements. The model elements are instances of the role's base.

Role Model : A Role Model characterizes a family of UML models. It specifies a structure of UML model elements in terms of roles and their relationships.

This paper describes a type of Role Model called a *Static Role Model* (SRM). An SRM is a characterization of a family of UML static structural models.

3.1 Static Role Models (SRMs)

The base of a role in a SRM is a metamodel class whose instances are elements of UML static models, that is, a SRM role characterizes a set of UML static modeling constructs. For example, a SRM classifier role (i.e., a SRM role with the metamodel class *Classifier* as a base) defines properties that classifiers (e.g., classes, interfaces) must have if they are to realize the role, while a SRM relationship role defines properties that UML relationships (e.g., associations, generalizations) must have if they are to realize the role.

Classifier Role. Two types of properties can be specified in a SRM classifier role:

- ***Metamodel-level constraints*** are well-formedness rules for the elements characterized by the role. The UML well-formedness rules and the metamodel-level constraints defined in a SRM role determine the form

of model constructs that can play the SRM role. Like the UML metamodel well-formedness rules, metamodel-level constraints are expressed using the Object Constraint Language (OCL) [OMG2001].

- ***Feature roles*** are characterizations of application-specific properties. A feature role consists of a name, a *realization multiplicity*, and an optional property specification expressed as a *constraint template*. Substituting parameters in constraint templates produces application-specific properties, called *model-level constraints*. A realization multiplicity specifies the number of realizations a feature role can have in a single realization of the enclosing SRM role. In this paper, we do not show feature role realization multiplicities if it is "1..*" (a commonly occurring case in the patterns we specified). The constraint template of a feature role determines a family of application-specific properties. Features (e.g., attributes, operations) of model constructs that play a SRM role realize feature roles of the SRM role. There are two types of feature roles:

  – ***Structural roles*** specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization. An example of a structural role that can be realized by class attributes is given below (see Fig. 1(a)):

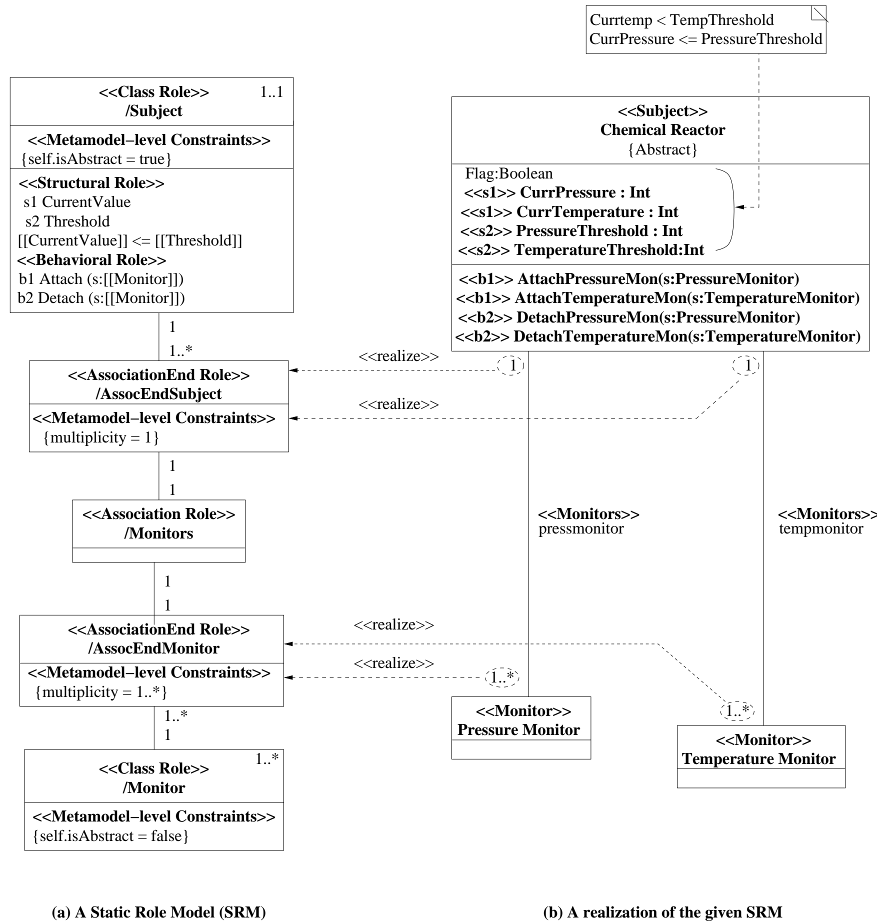
  ```
  CurrentValue {[[CurrentValue]] <= [[Threshold]]}
  ```

  In the above, *CurrentValue* is a feature role with an implicit realization multiplicity of 1..∗ indicating that there must be at least one realization of this role in a realization of the SRM role containing this feature role. The constraint template enclosed in the brackets {, } (parameters are surrounded by [[,]]) states that realizations of the *CurrentValue* feature role must always have a value less than the value of a realization of another feature role named *Threshold*.

  – ***Behavioral roles*** specify behaviors that are realized by a single operation or method, or by a composition of operations or methods in a SRM role realization. An example of a behavioral role is given below (see Fig. 1(a)):

  ```
  Attach (m:[[Monitor]])
  {pre:[[Monitor]] -> excludes(m)
  post:[[Monitor]] = [[Monitor]]@pre -> including(m)}
  ```

  The implicit realization multiplicity 1..∗ indicates that there must be at least one behavior that realizes this role in a realization of the SRM role containing this feature role. The specification of the behavior references a value m that is an instance of a realization of a SRM role called *Monitor*. The constraint template consists



(a) A Static Role Model (SRM) **(b) A realization of the given SRM**

Figure 1: An SRM for a Simple Pattern

of pre- and post-condition templates. The pre-condition template states that before execution of a behavior realizing *Attach*, m is not in the set of objects (instances of an *Monitor* realization) known to the object that performs the *Attach* behavior. The post-condition template states that the effect of a realization of *Attach* is to include m in the set of objects known to the performing object. The expression x@*pre* in the OCL refers to the value of x before execution of an operation.

Substituting the names of realizations for the feature or SRM role names enclosed in the double square brackets ([[,]]) results in an

application-specific property expressed in the OCL. For example, substituting *Sensor* (a realization of *Monitor*) for *Monitor* in the *Attach* constraint template results in the following model-level constraint:

```
Attach(m:Sensor)
  {pre:Sensor -> excludes(m)
   post: Sensor = Sensor@pre -> including(m)}
```

Establishing that a model element realizes a SRM role involves proving that the constraints associated with the model element imply the model-level constraints obtained by suitably instantiating the role's constraint templates, and determining that the realization multiplicities associated with the feature roles are not violated.

Relationship Role. A role can be associated with another role, indicating that the realizations of the roles are associated in a manner that is consistent with how the bases of the roles are associated in the UML metamodel. For example, a *Class* role can be directly associated with an *AssociationEnd* role, but not with an *Association* role because the UML metamodel does not directly associate the *Association* metamodel element with the *Class* metamodel element. We use the UML form of association to represent role associations. Like UML associations, role associations can be named and can have multiplicities associated with their ends. Any UML relationship (including n-ary associations) can be characterized by relationship roles.

A simple example of a SRM is given in Fig. 1(a). The SRM consists of two class roles, *Subject* and *Monitor*, an association role, *Monitors*, and two association end roles, *AssocEndSubject* and *AssocEndMonitor*. The *Monitor* role characterizes classes of objects (monitors) that observe other objects called subjects, and the *Subject* role characterizes subject classes. The association between subject and monitor classes in a realization of the pattern is captured in the pattern by *Monitors*. The form of the association ends of a realization of *Monitors* is constrained by the two association end roles shown. *AssocEndSubject* states that the subject end of a *Monitors* realization must have a multiplicity of 1. Similarly, *AssocEndMonitor* states that the monitor end of a *Monitors* realization must have a multiplicity of one or more. The SRM thus characterizes models in which subject objects can attach themselves to one or more monitor objects, and an monitor object can observe only one subject object.

The multiplicities on the association between the *Subject* role and the *AssocEndSubject* role indicate that a realization of the *Subject* role (a class) must have at least one association end that realizes *AssocEndSub*, and that a realization of *AssocEndSub* must be associated with exactly one class that realizes the *Subject* role. The other role associations shown in the SRM are

interpreted in a similar manner. The realization multiplicities shown in the SRM require that a realization of the SRM must have exactly one realization of *Subject* and at least one realization of *Monitor*, and a realization of *Subject* is associated with at least one *Monitor*, and each *Monitor* realization is associated with exactly one *Subject* realization via *Monitors* realizations. The smallest models that can realize this SRM consist of a realization of *Subject* and a realization of *Monitor* connected by a realization of *Monitors*.

Realizations of the *Monitor* role must be concrete classes ($self.isAbstract = false$). Realizations of the *Subject* role must:

- be abstract classes ($self.isAbstract = true$),
- have at least one realization of *CurrentValue* and *Threshold* and
- have behaviors that realize *Attach* and *Detach* feature roles. Pre- and Post- conditions for *Detach* behavior are given below:

```
Detach (m:[[Monitor]])
  {pre: [[Monitor]] -> includes(m)
   post: [[Monitor]] = [[Monitor]]@pre -> excluding(m)}
```

The behavioral model-level constraints that must be satisfied by realizations of *Subject* are obtained by instantiating the parameters of the behavioral roles constraint templates (i.e., *Attach*, *Detach*). A realization of the *Attach* behavior attaches a monitor to the subject, and a realization of the *Detach* behavior removes a monitor from the subject.

3.2 Realization Example

A realization of the simple SRM is shown in Fig. 1(b). In this paper, a stereotype with a role name (e.g., $\ll$ *Subject* $\gg$) in a realization is used to indicate that the model construct is an intended realization of the role. These stereotypes are printed in bold to distinguish them from other UML- and user-defined stereotypes.

The class *ChemicalReactor* is a realization of the *Subject* role, while *PressureMonitor* and *TemperatureMonitor* are realizations of the *Monitor* role, as indicated by the stereotypes. The attributes *CurrPressure* and *CurrTemperature* each play the role of *CurrentValue* indicated by $\ll s1 \gg$. Similarly, the attributes *PressureThreshold* and *TemperatureThreshold* each play the role of *Threshold* indicated by $\ll s2 \gg$. The *AttachPressureMon* and *AttachTemperatureMon* operations are intended to realize the *Attach* role and the *DetachPressureMon* and *DetachTemperatureMon* are intended to realize the *Detach* role. The behaviors specified by the model-level constraints obtained by appropriately instantiating the parameters of the *Attach* and *Detach* constraint templates are implied when *TemperatureMonitor* or *PressureMonitor* objects are passed as parameters to both the operations.

3.3 Role Models and the UML MetaModel

A Role Model determines a specialization of the UML metamodel in the sense that realizations of the Role Model are a well-defined subset of metamodel instantiations.

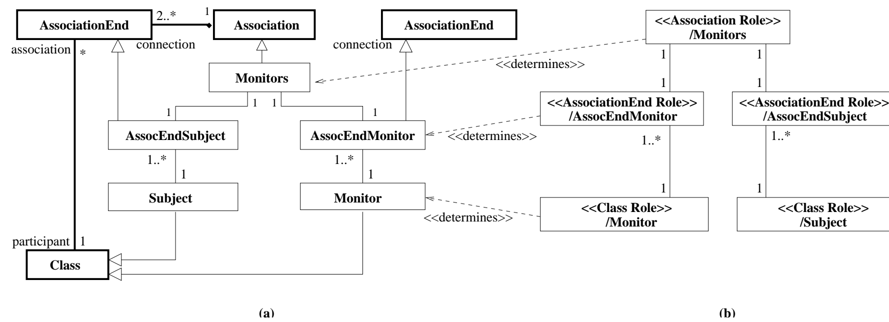


Figure 2: UML Metamodel view of the simplified Observer pattern SRM

Fig. 2(a) shows the metamodel and Fig. 2(b) shows a SRM for the simple pattern shown in Fig. 1. Classes that can play the *Subject* role are instances of the metamodel *Class* specialization called *Subject*, and classes that can play the *Monitor* role are instances of the metamodel *Class* specialization called *Monitor*. Similarly, the associations that can play the *Monitors* role are instances of a specialization of the *Association* metaclass called *Monitors*.

3.4 Role Model Abstraction

SRMs can be presented at various levels of abstraction. Fig. 3 illustrates the different forms of abstractions that we have defined for SRMs.

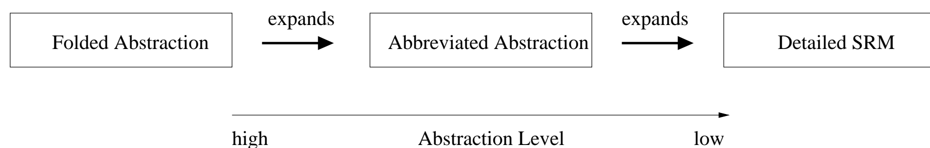


Figure 3: Role Model Abstraction

Folded Abstraction. SRMs often contain recurring structures that can be viewed as a pattern. An example of such a structure is the hierarchy structure shown in Fig. 6. The notion of hierarchy can be abstracted by a stereotype

structure ≪ *Hierarchy* ≫ to obtain an abstract view of a Role Model. Properties not shown at this level are: structures of hierarchies, metamodel-level constraints, and feature roles. An example of a folded abstraction is shown in Fig. 5.

Abbreviated Abstraction. Abbreviated SRMs expand (*unfold*) Folded SRMs. The expansion includes feature roles and hierarchies that were folded. SRMs are abbreviated in the sense that they still do not show details of metamodel-level constraints and model-level constraint templates. Abbreviated SRMs also hide details of relationship roles such as *AssociationEnd*, *Generalization*, and *Realization*. An example of an abbreviated abstraction is shown in Fig. 6.

Detailed SRM. A detailed SRM is a full specification of properties. An example is shown in Fig. 7.

4 PATTERN SPECIFICATION

In this section, we illustrate our technique for specifying patterns using the Abstract Factory pattern.

4.1 Abstract Factory Pattern Specification

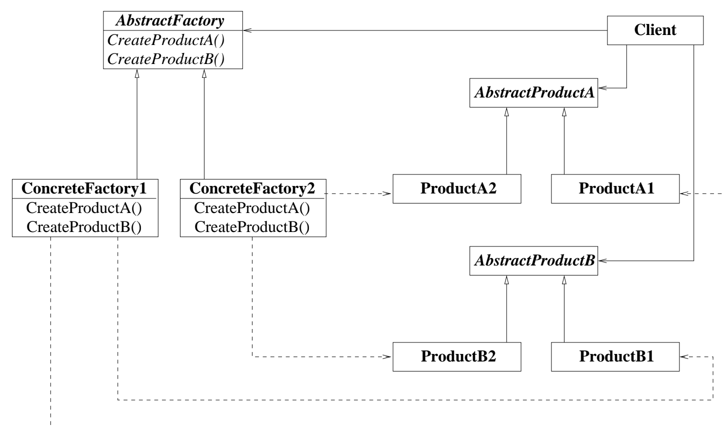


Figure 4: The GoF Abstract Factory Pattern

Fig. 4 shows the Abstract Factory (AF) pattern structure as presented in [GHJV1995]. A problem with the GoF description is that it is presented in terms of a typical realization of the pattern. Its use as a compliance point

against which proposed pattern realizations are checked is very limited. For example, the *AbstractFactory* depicted in the diagram is an abstract class, but the example provided in [GHJV1995] has an *AbstractFactory* realization that is a concrete class (the concrete operations provide default implementations). Inferring properties common to all pattern realizations from "typical" examples can result in unnecessary restrictions on what can be considered to be pattern realizations. Examples of what we consider to be unnecessarily restrictive pattern realization properties that one may infer from the GoF example are given below:

- *There must be one or more product specialization hierarchies with abstract root superclasses.* This unnecessarily precludes realizations that have (1) concrete root superclasses (to provide default implementations), (2) product hierarchies that are based on ≪ *realize* ≫ dependency relationships between interfaces or types and implementation classes, and (3) product classes that are not part of any specialization hierarchies.

- *Each concrete factory must have exactly one create operation for each product hierarchy.* This precludes realizations in which creation of certain products is optional, and realizations in which one or more specializations from the same product hierarchy can be used to build a structure.

The Role Models described in Section 3 can provide better compliance points against which proposed realizations can be checked because they precisely characterize realizations. In the following paragraphs, we show AF SRMs at different levels of abstraction.

Folded Abstract Factory SRM. Fig. 5 shows a Folded AF SRM that includes *Factory* and *Product* role hierarchies and *Client* role with two association roles (*ClientFactoryAccoc*, *ClientProductAssoc*) and two dependency roles (*ClientFactoryDep*, *ClientProductDep*).

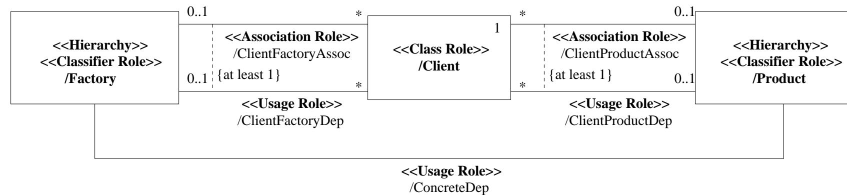


Figure 5: Folded Abstract Factory SRM

The multiplicity on *Client* restricts a realization of the AF SRM to consist of exactly one realization of *Client*. The pre-defined constraint {*at least* 1} shown between the *ClientFactoryAssoc* and *ClientFactoryDep* relationship roles

states that each pair of *Factory* and *Client* realizations must be connected by at least one relationship that is either an association (a realization of *ClientFactoryAssoc*) or a usage dependency (a realization of *ClientFactoryDep*). Similarly, a *Client* realization is connected to each *Product* realization via an association or a usage dependency.

Abbreviated Abstract Factory SRM. An abbreviated SRM of the Abstract Factory pattern is shown in Fig. 6. The figure shows unfolded role hierarchies of *Factory* and *Product* and their relationships.

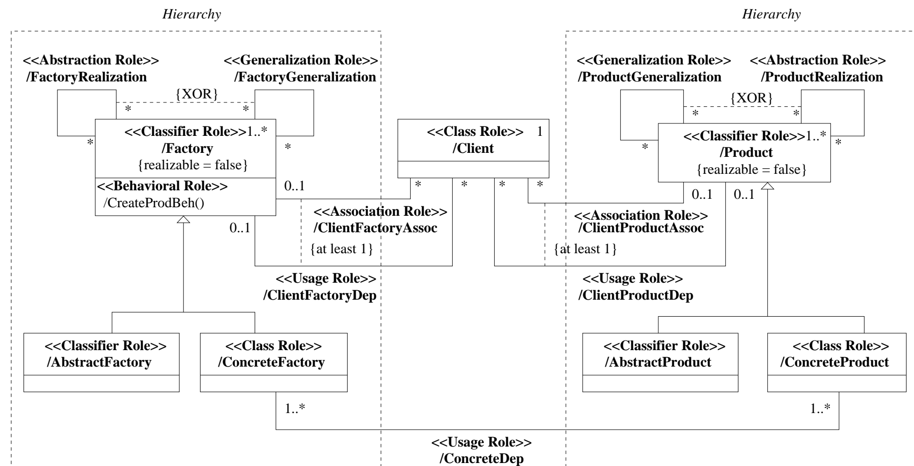


Figure 6: Abbreviated Abstract Factory SRM

The AF SRM consists of two major role structures: the *Factory* and the *Product* role hierarchies where the *Factory* and the *Product* roles are abstract (i.e., {realizable = false}). There are two types of *Factory* roles: *AbstractFactory* and *ConcreteFactory*. Realizations of the *Factory* specialization roles can be connected either by a realization of the *FactoryRealization* role (a UML ≪ *realize* ≫ dependency) or a realization of the *FactoryGeneralization* role (a UML generalization relationship). This role structure permits realizations that are (1) hierarchies of factory classifiers (based on generalization and ≪ *realize* ≫ relationships), and (2) classifiers that are not part of any hierarchy. Similarly, the Product role structure permits realizations that are hierarchies of product classifiers and classifiers that do not belong to any hierarchy.

Detailed Abstract Factory SRM. Fig. 7 shows a detailed view of the AF SRM (the dependency roles between *Client* and *Factory*, and between *Client* and *Product* are omitted in Fig. 7 to reduce clutter in the diagram).

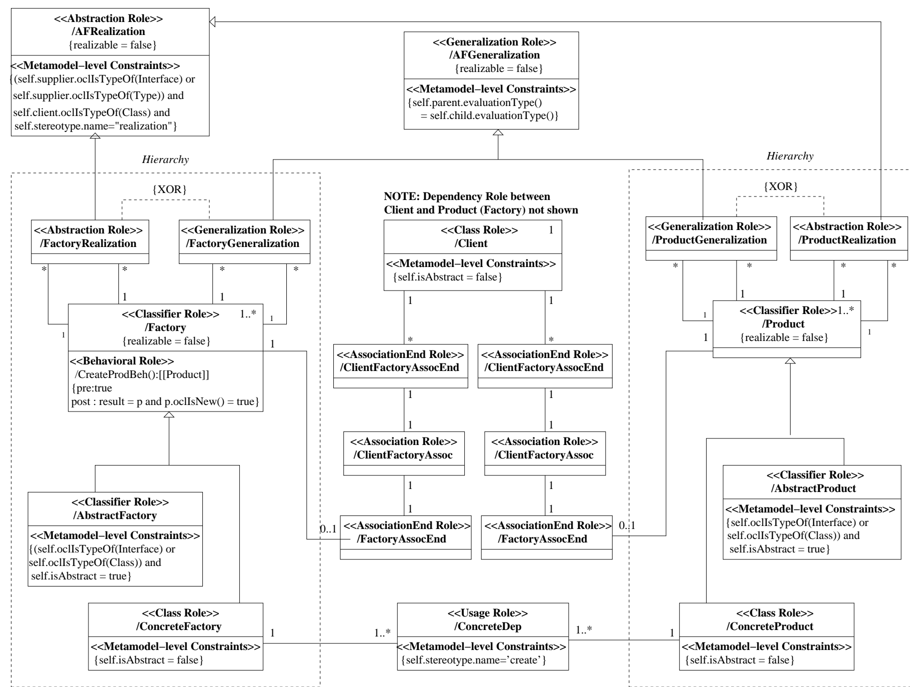


Figure 7: Detailed Abstract Factory SRM

The properties expressed in Fig. 7 are summarized below:

Factory. There must be at least one *Factory* realization in an AF SRM. Each *Factory* realization must have at least one relationship (an association or a dependency relationship) with a *Client* realization. *AbstractFactory* and *ConcreteFactory* realizations have the following properties.

- They have one or more behaviors that realize *CreateProdBeh*. Each behavior creates a new instance of a realization of *Product*.

```
CreateProdBeh():[[Product]]
{pre: true
 post: result=p and p.oclIsNew=true}
```

- They have 0 or more generalizations, or 0 or more ≪ *realize* ≫ relationships with other *Factory* realizations. Two *Factory* realizations cannot be connected by both a *FactoryRealization* realization and a *FactoryGeneralization* realization, as indicated by the {*XOR*} constraint between the

roles. In a ≪ *realize* ≫ relationship, the supplier must be an interface or a class that can have operations (but no methods), while the client must be a class. In a generalization relationship, the superclass and the subclass must both be interfaces or both be classes.

- *AbstractFactory* realizations can be abstract classes or interfaces, while *ConcreteFactory* realizations must be concrete classes.
- Realizations of *ConcreteFactory* are connected to one or more realizations of *ConcreteProduct* via usage dependencies (realizations of *ConcreteDep*).

Product. There must be at least one *Product* realization in an AF SRM. Each *Product* realization must have at least one relationship (an association or a dependency relationship) with a *Client* realization. *AbstractProduct* and *ConcreteProduct* realizations must have 0 or more generalizations, or 0 or more ≪ *realize* ≫ relationships with other *Product* realizations. Like *Factory* realizations, two *Product* realizations cannot be connected by both a *ProductRealization* realization and a *ProductGeneralization* realization. The generalization and ≪ *realize* ≫ relationship roles have properties similar to those in the *Factory* role structure. *AbstractProduct* realizations must be interfaces or abstract classes, while *ConcreteProduct* realizations must be concrete classes.

Client. A realization of *Client* must be a concrete class. *Factory* and *Product* realizations are connected to *Client* realizations via usage dependencies (not shown in Fig. 7) or associations.

The smallest models that can realize our AF SRM consist of a *Factory* realization (i.e., a realization of *AbstractFactory* or *ConcreteFactory*), a *Product* realization, a *Client* realization, a relationship between the *Client* realization and the *Factory* realization that is either an association or a usage dependency, and a relationship between the *Client* realization and the *Product* realization that is either an association or a usage dependency. If the *Factory* and *Product* realizations are concrete classes, they are connected by a usage dependency (a realization of *ConcreteDep*). Realizations that consist only of abstract classes or interfaces are permitted by the AF SRM. This permits the use of patterns in the development of design frameworks based on abstract classes.

5 ROLE MODEL SPECIALIZATION

Given a Role Model, one can loosen the constraints to include more models as realizations (Role Model generalization) or tighten the constraints to exclude some models as realizations (Role Model specialization). It may be possible in some cases to specialize a Role Model to the point that variations in the designs can be expressed as parameters. The resulting parameterized Role Models pave

the way for automated generation of initial designs from patterns. A generalization/specialization hierarchy of Role Models for a pattern can be beneficial in that it provides developers with a means for navigating to a Role Model that characterizes the smallest subset of realizations that have the desired properties. A Role Model is a specialization (child) of another (parent) Role Model if it further restricts the properties specified in the parent Role Model. A Role Model specialization characterizes a subset of its parent's realizations.

In this paper we focus only on SRM specialization. A SRM can be specialized by (1) specializing SRM roles, (2) further restricting the multiplicities on role associations, (3) reducing the number of alternatives by removing alternative structures and tightening the constraints (e.g., association and realization multiplicities) to allow only the remaining alternatives (e.g., a SRM that has generalization and $\ll$ *realize* $\gg$ relationship roles as alternative relationships can be specialized by removing the alternative generalization relationship role and allowing only $\ll$ *realize* $\gg$ relationships), and by (4) adding new roles and associations to the SRM that must be realized (i.e., requiring additional structure in realizations).

Fig. 8 shows two SRMs that are specializations of the abbreviated Abstract Factory SRM shown in Fig. 6. Fig. 8(a) is an Abstract Factory specialization in which:

- realizations of *AbstractFactory* are restricted to UML interfaces, and realizations of *Product* are restricted to classes,
- the pattern relationships between realizations of *AbstractFactory* and *ConcreteFactory* are restricted to UML $\ll$ *realize* $\gg$ dependencies (realizations of *FactoryRealization*), while the pattern relationships between realizations of *Product* are restricted to realizations of *ProductGeneralization*,
- the pattern relationship between *Client* and *Product* realizations are restricted to realizations of *ClientProductAssoc*, in which a *Client* realization must be associated with at least one *Product* realization and a *Product* realization is associated with at most one *Client* realization, and
- the pattern relationship between *Client* and *Factory* are restricted to realizations of *ClientFactoryAssoc*, in which a *Client* realization must be associated with at least one *Factory* realization, and a *Factory* realization is associated with at most one *Client* realization.

The specialization shown in Fig. 8(b) characterizes SRM realizations in which realizations of *Factory* and *Product* roles are classes. Hierarchies of *Product* and *Factory* realizations are formed using generalization relationships (realizations of *FactoryGeneralization* and *ProductGeneralization*). The other aspects of the pattern specialization are similar to the specialization shown in Fig. 8(a).

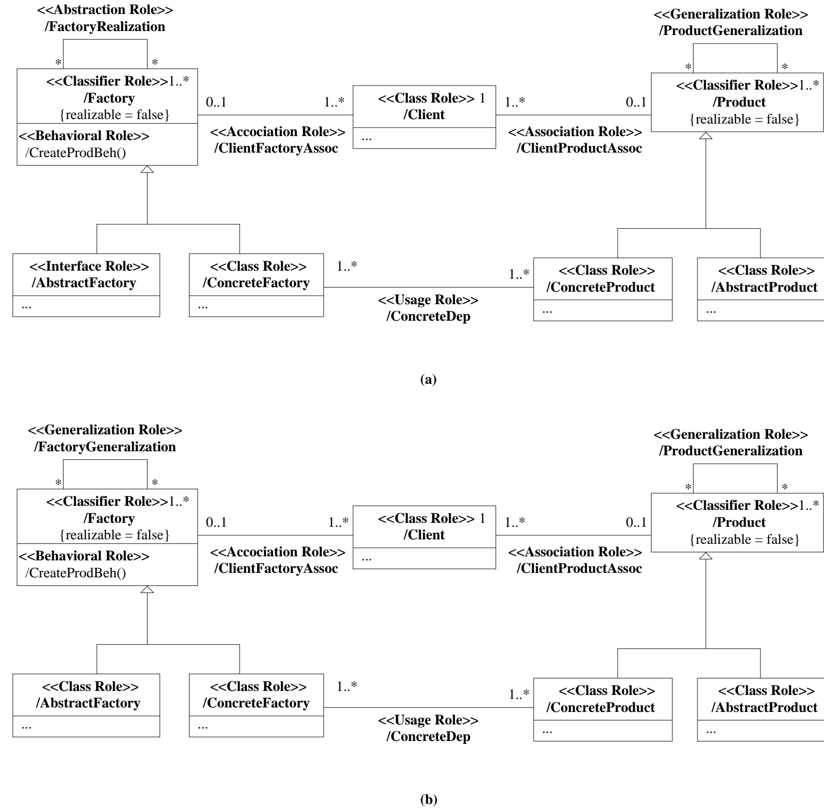


Figure 8: Specialized Abstract Factory Role Models

6 CONCLUSION AND FUTURE WORK

In this paper we introduce the notion of Static Role Models (SRMs) as a means for expressing pattern specifications. SRMs are based on a well-defined notion of a role, where a role is a specification that determines a subset of the instances of its base metamodel element. We are currently developing other types of role models that capture more complex behavioral constraints. Our current focus is on developing Interaction Role Models (IRMs) to characterize a family of interaction diagrams (e.g., collaboration and sequence diagrams) and State Chart Role Models to characterize a family of state chart models. We are also developing transformation techniques that use Role Models to support pattern-based refactoring of models.

REFERENCES

[Eden1999] A. Eden. Precise Specification of Design Patterns and Tool Support in Their Application. University of Tel Aviv, Israel, 1999

[GHJV1995] E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995

[GSJ2000] A.L. Guennec and G. Sunye and J. Jezequel. Precise Modeling of Design Patterns. Proceedings of UML'00, 2000

[ISO/IEC1996] RM-ODP, International Standard Organisation (ISO), Information technology - Open Distributed Processing - Reference model: Foundations, ISO/IEC JTC1/SC07, 10746-2, ITU-T Recommendations X.902 (1996)

[ISO/IEC2001] RM-ODP , International Standard Organisation (ISO), Information technology - Open Distributed Processing - Reference model: Enterprise Language, ISO/IEC ISO/IEC, 15414, ITU-T Recommendations X.911, Amendment1: Additional text (2001)

[LK1998] A. Lauder and S.Kent. Precise Visual Specification of Design Patterns. Proceedings of ECOOP'98, 1998

[OMG2001] The Object Management Group (OMG). Unified Modeling Language. OMG, http://www.omg.org, Version 1.4, September, 2001

[Riehle1998] D. Riehle and T. Gross. Role Model Based Framework Design and Integration. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98), 1998