

# Model Composition - A Signature based approach

Raghu Reddy, Robert France, Sudipto  
Ghosh  
Colorado State University  
601 S. Howes St.  
Fort Collins, CO 80523, USA  
{raghu,france,ghosh}@cs.colostate.edu

Franck Fleuery, Benoit Baudry  
IRISA  
Campus Universitaire de Beaulieu  
35042 Rennes Cedex, France  
ffleuery@gmail.com, bbaudry@irisa.fr

## ABSTRACT

The aspect oriented modeling (AOM) approach provides mechanisms for separating crosscutting functionality from other functionality. The AOM models crosscutting functionality as aspects and the business functionality as primary model. The overall system view is obtained by composing the primary and aspect models. In this paper, we present a model composition technique that relies syntactic pattern matching. A model is said to match with another if their signatures match. A signature consists of some or all properties of an element as defined in the UML metamodel. The technique proposed in this paper can be used to detect conflicts that arise during composition.

## Keywords

Aspect oriented modeling, composition, conflicts, signature, UML

## 1. INTRODUCTION

Modern software systems are complex. One of major factors of software complexity is the need for system developers to address multiple crosscutting features (e.g., access control, availability, and error recovery). If the functionality that addresses these design features is spread across and intertwined with design elements that address other business features, then the features are said to be crosscutting. The Aspect-oriented modeling (AOM) approach provides mechanisms for separation of such crosscutting functionality from business functionality.

In the AOM approach, features that crosscut the modules of a primary (business) model are described separately in aspect models. The aspect and primary models are described using the Unified Modeling Language (UML)[16]. Composition of aspect models and a primary model yields an integrated design model [5, 13] that reflects the overall system view. Composition is necessary to identify conflicts

that may arise as a result of interactions among aspect and primary model elements.

The composition procedure used in AOM is name based. For example, if two elements are of the same name, they will be merged. This may give rise to conflicts. One example of such conflict is when the two elements have the same name but may not be of the same element type. Another conflict may be when the elements have the same name but the meta-class of the element may not be the same. Such conflicts can be detected by using a signature-based composition approach instead of a name based approach. Composition directives [15] can be used to resolve some of these conflicts involving undesirable emergent behavior. Please see [15] for more details on composition directives. In this paper, we provide a signature based technique that uses signatures, rather than names, to determine matching model elements.

The composition approach provided in this paper is different from our previous work [5, 12, 14] in the following: (1) we introduce a more general form of model element matching that is based on the notion of model element signatures, (2) we propose a composition metamodel that describes static and behavioral properties of composable model elements. The composition metamodel includes elements from the UML metamodel, but unlike the UML metamodel, the composition metamodel contains descriptions of behavior. The metamodel is intended to guide the development of model composition tools. (3) we provide a brief outline of the algorithm being implemented for model composition using kermeta [?].

The rest of the paper is organized as follows. Section provides a brief overview of our aspect oriented modeling approach. Section 3 describes the composition meta-model based on signatures. Section 4 outlines the algorithm for signature-based model composition. Section 5 illustrates the composition approach using a simple example. Section 6 presents some related work and section 7 presents the conclusions and future work.

## 2. BACKGROUND

### 2.1 Aspect Oriented Modeling

In the AOM approach, aspect models are generic (parameterized) descriptions of a crosscutting feature. The aspects models consist of structural and behavioral UML template diagrams [5]. The template notation is adapted from our pattern specification language called Role-Based Metamodeling language (RBML)[4]. An aspect model must

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

be instantiated before it can be composed with a primary model. The instantiated forms of aspect models are referred to as *context-specific aspects*. The instantiation of an aspect model is determined by *bindings*, where a binding associates a value representing an application-specific concept with a template parameter. Composition directives[?]. Composing context-specific aspects and a primary model produces an integrated view of the design.

## 2.2 Essential Meta-Object Facilities

Essential Meta-Object Facilities (EMOF) 2.0 has been proposed by the Object Management Group [10]. EMOF is a minimal meta-modeling language designed to specify meta-models. It provides the set of elements required to model object oriented systems. The minimal set of EMOF constructs required for the composition algorithm, that are provided by EMOF are specified in figure 1.

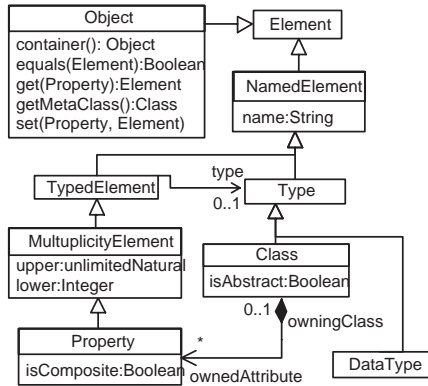


Figure 1: EMOF classes required for composition

All Objects have a class which describe its properties and operations. An Object extends an Element. The `getMetaClass()` method returns the Class that describes this Object. The `container()` method returns the containing parent object. It returns null if there is no parent object. The `equals(Element)` determines if the element is equal to this object instance. The `set(Property, Element)` sets the value of the Property to the Element. The `get(Property)` returns a List or a single value depending on the multiplicity.

The `isComposite` attribute under class Property returns true if the object is contained by the parent object. Cyclic containment is not possible, i.e. an object can be contained by only one other object. The `getAllProperties()` method of the Class returns all the the properties of instances of this Class along with the inherited properties. The attribute `upper` and `lower` of class MultiplicityElement represents the multiplicities of the associations at the meta-model level. For example, “0..1” represents lower bound “0” and upper bound “1”. If the upper bound is greater than “1” then the property value is null or a single object otherwise its a collection of objects.

## 2.3 KerMeta

Kermeta[1] is an open source meta-modeling language developed by the Triskell team at IRISA. It has been designed as an extension to the EMOF 2.0[10]. Kermeta extends EMOF with an action language that allows specifying semantics and behavior of meta-models. The action language

is imperative and object-oriented. It is used to provide an implementation of operations defined in meta-models. A more detailed description of the language is presented in [9].

The Kermeta action language has been specially designed to process models. It includes both OO features and model specific features. Kermeta includes traditional OO static typing, multiple inheritance and behavior redefinition/selection with a late binding semantics. To make Kermeta suitable for model processing more specific concepts such as opposite properties (i.e. associations) and handling of object containment have been included. In addition to this, convenient constructions of the Object Constraint Language (OCL) such as closures (e.g. each, collect, select) are also available in Kermeta.

To implement the composition algorithm we have chosen to use Kermeta for several reasons. Firstly, the language allows implementing composition by adding the algorithm in the body of the operations defined in the composition metamodel. Secondly, it can be adapted to model processing and can include the reflection capabilities on meta-models that are required to implement the algorithm. Lastly, the Kermeta tools are compatible with the Eclipse Modeling Framework (EMF) which allows us to use Eclipse tools to edit, store and visualize models.

## 3. SIGNATURE BASED COMPOSITION

The composition procedure involves merging corresponding UML diagrams of context-specific aspect and primary models to produce a composed model using a name based approach. Conflicts and undesirable emergent behavior can arise as a result of composing the aspect and primary models. This can be identified during composition or during analysis of the composed model. For example, it may be the case that two classes with the same name may not represent the same concept or have conflicting properties. The use of an application domain namespace can help reduce the occurrences of some naming conflicts, but this is not enough. Signature-based composition technique helps in identifying conflicts explicitly during composition.

In signature-based composition, information in model elements with matching signatures are merged to form a single model element in the composed model. The *signature* of a model element is a set of property values, where the properties are a subset of properties (attributes and association ends) associated with the class of the model element in the UML metamodel. The set of properties used to determine a signature is called a *signature type*. For example, the signature type for an operation can be defined as a set consisting of the operation name and its sequence of parameters.

**Signature Type:** {operation (name, {parameter (name, type)} ) }

Using the signature type given, the signature for two update operations with different sets of parameters is written as:

**Signature:** {update, {(x, int) (y, int)}}}

**Signature:** {update, {(s, String)}}}

The properties that define that signature type can vary from being just the name of the elements to all the constituent properties associated with the element. In the example given above, curly braces represent that there may be a set of parameters. Parentheses represents that the properties are optional. If there are more properties associated with the signature type, the number of elements with matching signatures will likely be lesser. For example, the above signatures

would have been a match if the signature type was just the name of the operation. It will not be a match if the signature type is defined as operation name and its sequence of parameters.

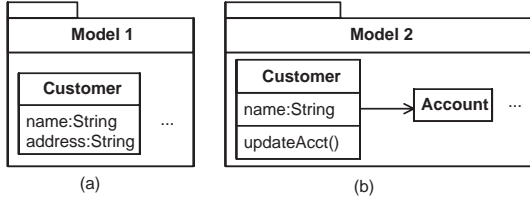


Figure 2: A simple merge example

Consider the simple example shown in figure 2 in which a model, contains a class named Customer with attributes name and address, (see Fig. 2(a)) and another model, contains a class named Customer with an attribute name and a reference to an Account object (see Fig. 2(b)). If the signature type for class is defined as consisting of the properties name, attributes and association ends, then the two classes do not match and thus are not merged. Note that, under the assumption that the signature accurately determines the classes that represent the same concept, this merge produces a faulty model: Two classes in a namespace have the same name but represent different concepts.

If the signature type consists only of the class name property then the two classes match and their contents are merged to form a single class. If a model element property is not included in a signature then it is subject to its own matching rules during the merge. So, care should be taken to specify the signature types for all model elements that need to be composed. In general, the following rules determine how properties in matching model elements are merged:

- If properties represented by model elements (e.g., class attributes) have matching signatures then they appear only once in the containing merged element.
- If a property in one matching element is not in the other, then it appears in the composed model element.

Using the above rules and signatures for attribute and association ends that require exact matches (i.e., each signature type contains all properties of the element type), merging of the two Customer classes in Fig. 2(a) and (b) produces a model with two classes Customer and Account. The Customer class will have two attributes name and address, an operation updateAcct() and a reference to the Account class.

## 4. COMPOSITION METAMODEL

The composition metamodel describes how signature based composition can be accomplished. The metamodel shown in figure describes the static and behavioral properties needed to support model composition in AOM. In this paper, we describe the behavioral properties in terms of class operations and narrative descriptions of the operations. Alternatively, sequence and activity diagrams can be used to describe the interactions and activities that take place during composition.

The core concepts of the composition metamodel shown in figure 3 can be used to compose any two models. In AOM,

the composition always starts with the primary model being the initial model. The aspect model are merged with the primary model. At any given time only one aspect model is composed with the primary model and the ordering is determined using the order of composition. The order of composition can be specified using composition directives.

The core concepts shown are described below:

- **Element:** Element is an extension of the UML meta-class Element. It is extended by adding the operational features *getMatchingElements(e[]: Element)*. Other methods *container()*, *get(property)*, *set(property,element)*, *getMetaClass()* of the EMOF Object class shown in figure 1 are used by the Element.
- **getMatchingElements:** This operation takes in a set of elements and returns an element or set of elements that have the same syntactic type and signature as the element that invokes it. The syntactic type check is performed by invoking the *getMetaClass()* and the *getAllProperties()* method of EMOF Object class.
- **Mergeable:** This is an abstract class. Instances of mergeable class are elements that are mergeable. Examples of mergeable elements shown in the figure are *Classifiers, Operations, and Models*.
- **merge:** This operation merges the element with another mergeable element.
- **sigEquals:** This operation checks if the element's signature is equal to the signature of another element.
- **getSignature:** This operation gets the signature of the element based on the signature type.
- **Signature:** This class is used to obtain the signature of the mergeable elements. This class is linked to every mergeable element.

The composition meta-model is primarily used for the development of model composition tool. We have implemented the model composition tool using kermeta. The composition metamodel classes were added to the kermeta metamodel and the operational features associated with it were implemented using kermeta.

## 5. COMPOSITION ALGORITHM

The composition algorithm takes in two models and outputs the resultant model. In AOM, primary model and aspect model as the two models. Composition yields a composed model that in turn will become the primary model for adding other aspects. The algorithm presented describes how two models can be composed based on signatures.

The algorithm assumes that the model elements are represented as objects (instances of EMOF::Object class). This is necessary because the algorithm is written independent of the model elements. The algorithm uses reflection to obtain structures of objects. For example, given the figure2a the algorithm takes a metamodel instance for all the elements specified. The metamodel instance diagram is shown in figure 5.

The models are merged only when the elements are of the same syntactic type and have same signature. The sigEquals() is shown as a pre-condition in figure 4. Each type of model element defines its own procedure for checking equality of signatures, that is, specializations of Mergeable can override the inherited sigEquals(). The models are merged by invoking the merge method. The merge method returns

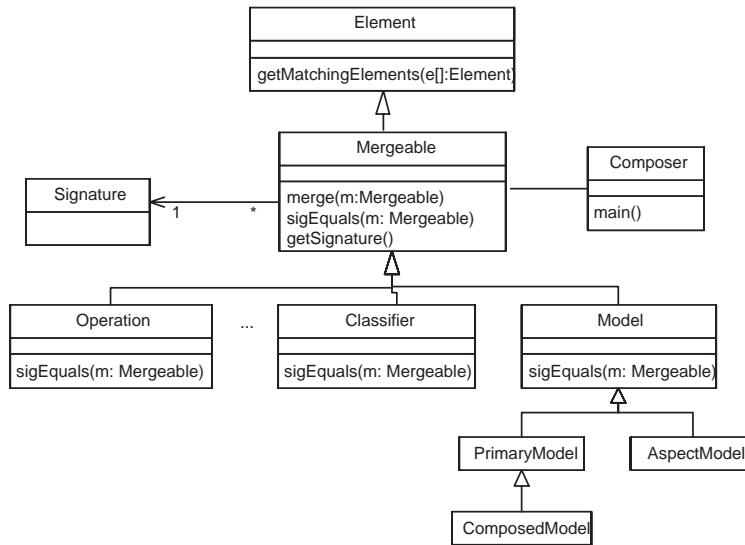


Figure 3: Composition metamodel

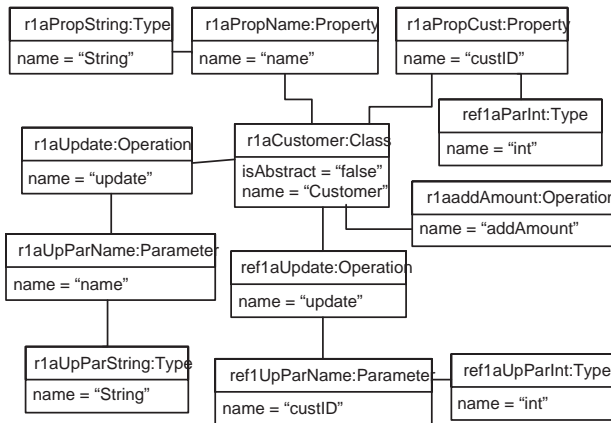


Figure 5: Metamodel Instance for the figure 2a

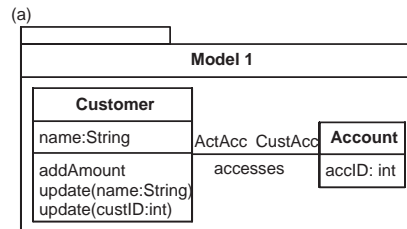
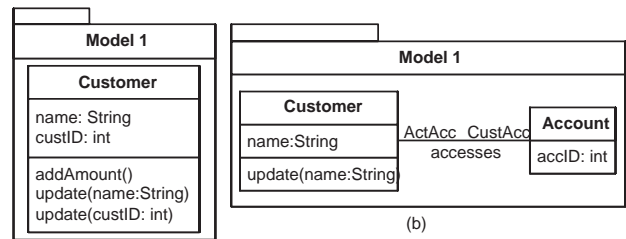
a new element that is the merge of mergeable element m and the element on which the merge is called.

The merge method in the algorithm proceeds as follows for all properties of the objects to merge:

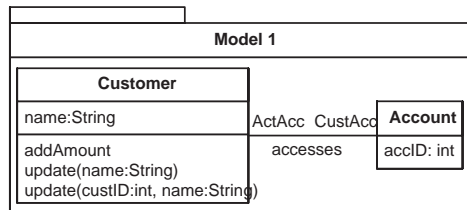
- If the property holds simple types then the property values should be the same, otherwise a conflict is detected. The conflict needs to be handled explicitly.
- If the property is a composite then the merge method is invoked recursively on all property values that have the same signature. The recursive merge call differs based on the upper bound of the property.
  - If the upper bound is equal to 1 the properties are merged depending on the signature. If the the signature does not match then a conflict is detected.
  - If the upper bound is greater than 1, then and each of the collection of object properties will be checked for matching element properties. If the

property values do not have the same signature they are added individually to the merged model element

## 5.1 Illustrative example



(c) Signature type is defined as the operation name and its parameter



(d) Signature type is defined as the operation name

Figure 6: A simple example illustrating composition variants

Consider the simple example shown in figure 6 [] in which

```

*****
Operation sigEquals determines if two model elements should be merged
e1.sigEquals(e2 : ModelElement)
*****
// Compare syntactic type and signature
return e1.getMetaClass == e2.getMetaClass and e1.getSignature == e2.getSignature

*****
// e1 and e2 are the model elements that need to be merged
e1.merge(e2 : ModelElement)
*****
precondition : e1.sigEquals(e2) returns true

// create the merged instance in the context of e1
result := e1.getMetaClass.new

// Iterate on all properties of the objects to be merge
// e1 and e2 have same meta-class. So they have same set of properties
foreach Property p in e1.getMetaClass.allProperties

  if type of p is primitive
  // Primitive type is the basic datatypes like string, int, etc,.
  // If an object does not have a value for a property then
  // the value val is taken from the other object and vice versa. This is not a conflict
  // If neither objects have values then val is null in the merged object.
  if e1.get(p) is null or e2.get(p) is null then
    result.set(p, val)
  else
    // if the values are the same then it is ok otherwise a conflict has been detected
    if e1.get(p) = e2.get(p) then
      result.set(p, e1.get(p))
    else
      A conflict has been detected
  else
  // Type of p is not primitive.
  // If the property refers to a single object
  if the property upper bound is 1
    if e1.get(p) is null or e2.get(p) is null then
      result.set(p, val) // val is the same as above
    else
      if sigEquals(e1.get(p), e2.get(p)) then
        // If the object e1.get(p) is contained by e1 and same for e2
        // (p.isComposite=true) then the objects should be merged, otherwise,
        // one is choosen.
        // Either one can be chosen because they both have the same signature
        if p.isComposite is true then
          result.set(p, merge(e1.get(p), e2.get(p)))
        else
          result.set(p, e1.get(p).clone())
      else
        A conflict has been detected
  else
  // The property refers to a collection of objects.
  // The merged object should contain property values that are only
  // in e1 or only in e2, and the merged version of objects that are in both e1 and e2.
  for each value v1 in e1.get(p)
    for each matching element v2 in e2.get(p)
      if p.isComposite then
        result.get(p).add(merge(v1, v2))
      else
        result.get(p).add(v1.clone())
        if no element found
          result.get(p).add(v1.clone())
  for each value v2 in e2.get(p)
    if NO matching element found in e1.get(p)
      result.get(p).add(v2.clone())

```

Figure 4: The merge part of composition algorithm

there are two packages with a model each. The first package contains Model 1 which contains a class named Customer with attributes *name* and, *cusID*, operations *addAmount*, *update* with a string parameter and, *update* with an integer parameter (see Fig. 6(a)). The second package contains a model, Model 1, which has a class named Customer and a class name Account. Customer class contains an attribute *name*, a reference to an Account object named *CustAcc*, and an operation named *update* with a string parameter. Account class contains an attribute *accID* and a reference to a Customer object named *ActAcc* (see Fig. 6(b)).

If the model composition algorithm is applied on the packages, the algorithm will compose the packages based on the signature type. If the signature type has been defined as model name, class name, operation name and, parameter name and type the algorithm will produce the composed model shown in Fig. 6(c). On the other hand if the signature type is defined as the model name, class name, operation name without the parameter name and type, the composition will produce the model shown in Fig. 6(d). There will be an explicit warning before such a merge is performed because the parameters do not match. The conflict can be resolved by continuing with the default merge which appends the parameters or by the developer in his or her own way.

The example given above is fairly intuitive and can be done manually. In cases where there are a large number of classes and associations, this becomes extremely complex. The tool can resolve the problem by composing the models automatically. The developer can input the signature type and the composition can be varied based on the signature type. Since the conflicts are detected during the composition itself, they can be handled the developer desires.

We have used to tool to compose fairly complex models. We have applied the tool on a partial banking application and composed it with a context-specific authorization aspect to obtain the composed model.

## 6. RELATED WORK

Grundy[7] proposes an Aspect Oriented Component Engineering (AOCE) approach that focuses on capturing concerns that crosscut many components. The approach is aimed at design and deployment level. The approach uses a set of UML meta-model extensions to make the aspect information explicit. The integrated view is a just a view that has the model elements referring to each other. Composition in AOCE occurs at run-time. Composition semantics are specific to the AOCE implementation platform.

The Aspect Modeling Language (AML) [6] is a UML based notation and is limited to AspectJ language constructs and the design notation described in UML For Aspects (UFA)[8]. Aspects are specified as packages of the <<aspect>> stereotype. Composition semantics follow the AspectJ implementation of AOP. Since, the AspectJ composition is limited in scope, the composition semantics for the approach are also limited.

In the approach proposed by Clarke *et al.*, [2, 3] a design, called a subject or theme, is created for each system requirement. A comprehensive design is created by composing all subjects; there is no notion of a primary design model separate from aspect models. Subjects are expressed as UML model views, and composition merges these views. Composition includes adding and overriding named elements in

a model. Conflict resolution mechanisms consist of defining precedence and override relationships between conflicting elements. Our composition procedure is more advanced in that we use signatures rather than names and also our the conflicts during composition can be explicitly handled.

As part of the early aspects initiative, Rashid *et al.* have targeted multi-dimensional separation throughout the software cycle [11]. Their work supports modularization of broadly scoped properties at the requirements level to establish early trade-offs, provide decision support and promote traceability to artifacts at later development stages. Our AOM approach complements their approach at the design level by providing mechanisms for composition and detecting conflicts.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we presented a composition technique for composing aspect and primary models that uses signature based mechanisms rather than name based mechanism. Composition of aspect models and a primary model may produce conflicts and undesirable emergent behavior and this can be explicit detected by the composition technique. The developer may choose the composition variants as desired.

We have developed a tool using kermeta to support the signature based composition technique. The inputs to the tool are aspect model and a primary model. The output is a composed model that can be used a primary model to compose with another aspect. We are currently in the process of adding composition directives that constrain how model elements are composed. The result of the signature-based composition is an integrated model that can be altered based on the set of composition directives.

We are also developing a prototype integrated toolset that supports the AOM approach. To date, our work on tool development has produced the following: (1) A prototype editor for creating aspect model class diagram templates, (2) A tool, built on top of Rational Rose, that generates instantiations from template forms of UML class diagrams (generic aspects) and (3) A model composer for composing the primary and context-specific aspect class diagrams.

## 8. REFERENCES

- [1] The KerMeta Project Home Page. *URL* <http://www.kermeta.org>.
- [2] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *The 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, 2001*.
- [3] S. Clarke and R. J. Walker. Towards a standard design language for AOSD. In *The 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April 2002*.
- [4] R. B. France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Trans. on Software Eng.*, 30(3):193–206, March 2004.
- [5] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *IEEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4):173–185, August 2004.
- [6] I. Groher and S. Schulze. Generating aspect code from uml models. In *Workshop on Aspect Oriented*

*Modelling with UML*, San Francisco, CA, October 2003.

- [7] J. C. Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 20(6), December 2000.
- [8] S. Herrmann. Composable designs with ufa. In *Workshop on Aspect Oriented Modelling with UML (held with AOSD 2002)*, Enshede, Netherlands, 2002.
- [9] P. Muller, F. Fleuery, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005 - to appear*, Montego Bay, Jamaica, October 2005.
- [10] OMG Adopted Specification ptc/03-10-04. The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, <http://www.omg.org>.
- [11] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *IEEE Joint Intl. Conference on Requirements Engineering*, pages 199–202, Essen, Germany, September 2002.
- [12] R. Reddy, R. B. France, and G. Georg. Aspect oriented modeling approach to analyzing dependability features. In *Aspect Oriented Modeling workshop held with Aspect Oriented Software Development conference*, Chicago, March 2005.
- [13] Y. R. Reddy, R. B. France, and G. Georg. An aspect-based approach to modeling and analyzing dependability features. Technical Report CS04 - 109, Colorado State University, November 2004.
- [14] E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control features and applications. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, Scandic Hasselbacken, Stockholm, June 2005.
- [15] G. Straw, G. Georg, E. Song, S. Ghosh, R. B. France, and J. Bieman. Model composition directives. In *Seventh Intl. Conference on the UML Modeling Languages and Applications*, Lisbon, Portugal, October. Springer.
- [16] The Object Management Group. Unified Modeling Language: Superstructure. Version 2.0, OMG, ptc/03-07-06, 2003.