Model Composition - A Signature-Based Approach

Raghu Reddy, Robert France, Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, CO, USA

Franck Fleurey, Benoit Baudry
IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France

E-mail: `raghu@cs.colostate.edu`

Abstract

The aspect oriented modeling (AOM) approach provides mechanisms for separating crosscutting functionality from core functionality in design models. Crosscutting functionality is described by aspect models and the core application functionality is described by a primary model. The integrated system view is obtained by composing the primary and aspect models. In this paper, we present a model composition technique that relies on signature matching: A model element is merged 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 is capable of detecting some conflicts that can arise during composition.

Keywords

Aspect oriented modeling, Composition, Conflicts, EMOF, KerMeta, Signature, UML

1. Introduction

A major factor behind the complexity of developing dependable software is the need to incorporate multiple interdependent features in a design (e.g., access control, availability, and error recovery). In this paper, a design feature is a logical unit of behavior, i.e., it is a piece of functionality. A design feature that has elements that are spread across a design and intertwined with elements from other features is called a crosscutting feature. The Aspect-oriented modeling (AOM) approach provides mechanisms for separation of crosscutting features from other features [5, 12].

In the AOM approach, core application features are described in a primary model and features that crosscut core application features are described by aspect models. Aspect and primary models are described using the Unified Modeling Language (UML)[14]. Composition of aspect models and a primary model yields an integrated design model. Composition is necessary to identify conflicts that may arise as a result of interactions between aspect and primary model elements.

The composition procedure used in our previous AOM approach was name-based. Name-based composition can give rise to conflicts in the cases where the element name is not enough to identify matching concepts. For example, when two attributes from different models are of the same name, they will be merged even if one attribute is of *string* type and another of *int* type. Merging these elements gives rise to a conflict because they are not of the same type. Composition directives [13] were used to resolve some of these conflicts. A composition directive specifies how to compose aspect models with a primary model. In this paper, we provide a signature-based composition technique that can be used to avoid some of the conflicts that can arise from using only name-based matching.

A signature consists of some or all of the properties associated with an element in the UML metamodel. For example, the signature of an attribute can be defined as consisting of its name and type. The developer can declare the signatures type. The defualt signature is just the name of the model element. In signature-based matching, elements of a type with matching signatures represent different views of the same concept and are thus merged in the composed model. This new approach does not remove the need for composition directives but it reduces the composition cases that require their use.

In this paper we present the following: a more general form of model element matching based on model element signatures, a composition metamodel that describes static and be-

havioral properties of composable model elements, an algorithm for composing two models and, a KerMeta [1] implementation of the composition algorithm.

The rest of the paper is organized as follows. Section 2 provides some background information. Section 3 gives an overview of signature based composition. Section 4 describes the composition meta-model based on signatures. Section 5 outlines the algorithm for signature-based model composition and provides a simple illustrative example. Section 6 presents some related work and section 7 presents the conclusions and future work.

2. Background

This section provides an overview of the aspect oriented modeling (AOM) approach, extended metamodeling facilities (EMOF), and KerMeta. We implement the new signature based AOM approach using KerMeta which manipulates metamodels written in EMOF.

2.1 Aspect Oriented Modeling

In the AOM approach, aspect models are pattern descriptions of crosscutting features. The aspects models consist of structural and behavioral UML template diagrams [5]. The template notation is a specialized form of pattern specification language called the Role-Based Metamodeling Language (RBML)[4]. An aspect model must 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 of an application-specific concept with a template parameter. Composition directives [13] can be used to influence how context-specific aspects are composed with the primary model. Composing context-specific aspects and a primary model produces an integrated view of the design.

2.2 Essential Meta-Object Faclitites

Essential Meta-Object Facilities (EMOF) 2.0 is a minimal meta-modeling language designed to specify meta-models [10]. It provides the set of elements required to model object oriented systems. The minimal set of EMOF constructs required for the composition algorithm is presented in Figure 1.

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

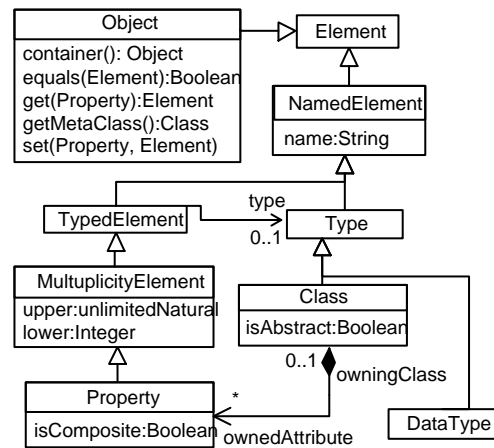The isComposite attribute under class Property returns true

Figure 1. EMOF classes required for composition

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()* operation (not shown in the figure) of the Class returns all the properties of instances of this Class along with the inherited properties. The attributes, upper and lower, of class MultiplicityElement, represent the mulplicities of the associations at the metamodel level. For example, "0..1" represents a lower bound "0" and an 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. 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 two reasons. First, the language allows implementing composition by adding the algorithm in the body

of the operations defined in the composition metamodel. Second, 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

In signature-based composition, information in model elements with matching signatures is 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 (e.g., 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 element to all the constituent properties associated with the element. In the example given above, curly braces indicate that there may be a set of elements that use the same signature type (operations and parameters in this case). Parentheses indicate 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 smaller. 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.

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)). Assuming the signatures match at the model level, if the signature type at the class level 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 (see Fig. 2(c)): 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 signatuare types for all model elements that need to be com-
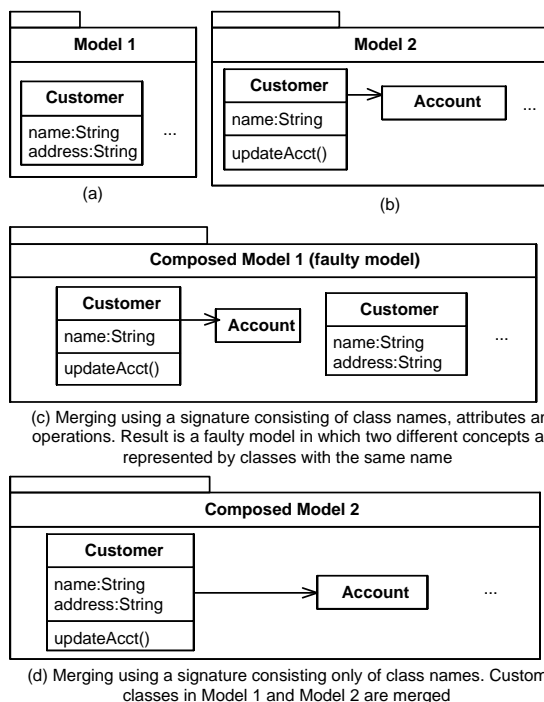

(a)

(b)

(c) Merging using a signature consisting of class names, attributes and operations. Result is a faulty model in which two different concepts are represented by classes with the same name

(d) Merging using a signature consisting only of class names. Customer classes in Model 1 and Model 2 are merged

Figure 2. A simple merge example

posed. 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 (see Fig. 2(d)).

4. Composition Metamodel

The composition metamodel describes how signature-based composition can be accomplished. The metamodel shown in Figure 3 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.

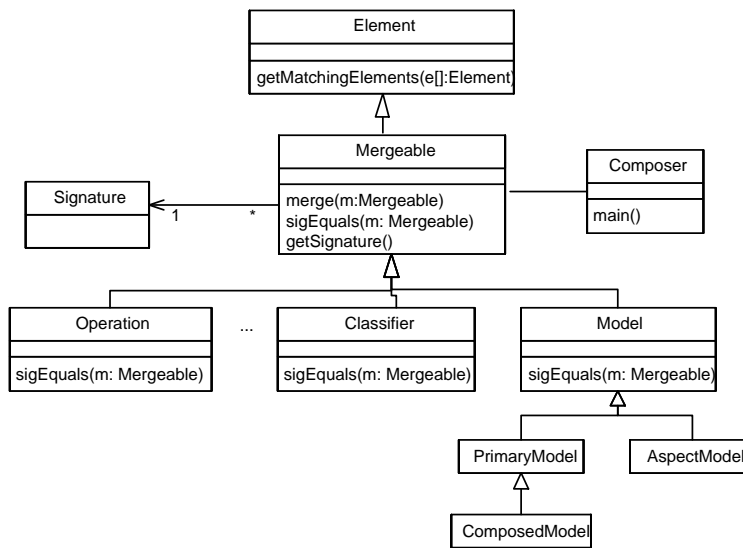The core concepts of the composition metamodel can be

Figure 3. Composition metamodel

used to compose any two models. In AOM, the composition always starts with the primary model being the initial model. The aspect model is merged with the primary model. At any given time, only one aspect model is composed with the primary model and the ordering can be changed using model composition directives.

The core concepts shown in Figure 3 are described below:

- **Element**: Element is an extension of the UML metaclass, Element. It is extended by adding the operations, *getMatchingElements(e[]: Element)*. Other operations *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 *getAllPropeties()* operation of EMOF Object class. The signature is obtained using the *getSignature()*.
- **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 a model composition tool. We have implemented the model composition technique using KerMeta. The composition metamodel classes were added to the UML 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 are the two models. The composition results in a composed model that in turn will become the primary model for adding other aspect models. The algorithm presented below 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 element types. The algorithm uses reflection to obtain structures of objects. For example, In Figure 2a the metaclass of Customer is Class and metaclass of update is Operation. The algorithm implicitly uses the metamodel instance for all the elements specified and then merges the model elements based on the metamodel instances. The metamodel instance diagram for Figure 2a is shown in Figure 4.

The models are merged only when the elements are of the same syntactic type and have the same signature. The *sigEquals()* operation is shown as a pre-condition to the merge method (see appendix). 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 (shown in the appendix) returns a new ele-
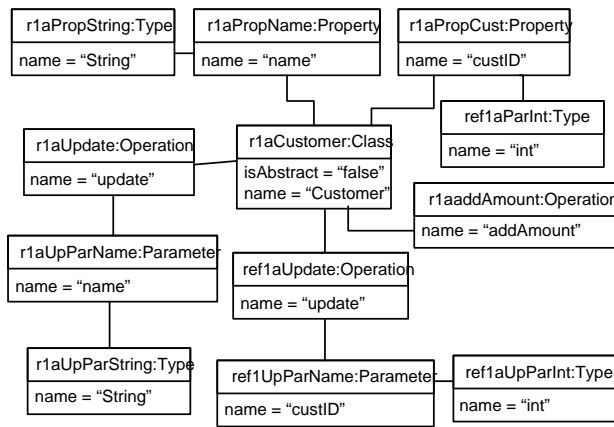
Figure 4. Metamodel Instance for the Figure 2a

ment 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 be merged:

- If the property is of simple type, then the property values should be the same; otherwise a conflict is detected. The conflict needs to be handled explicitly by the developer.
- If the property is composite, then the merge method is invoked recursively on all property values that have the same signature. The recursive merge call depends 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 signature does not match, then a conflict is detected.
  - If the upper bound is greater than 1, then each element 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

Consider the example shown in Figure 5 in which there are two packages each containing a model. The first package contains Model 1 which has a class named Customer, with attributes *name* and *cusID*, operations *addAmount*, *update* with a string parameter, and *update* with an integer parameter (see Fig. 5(a)). The second package contains a model, Model 1, which has a class named Customer and a class named Account. The Customer class contains an attribute, *name*, a reference to an Account object named *CustAcc*, and an operation named *update* with a string parameter. The Account class contains an attribute *accID* and a reference to a Customer object named *ActAcc* (see Fig. 5(b)).

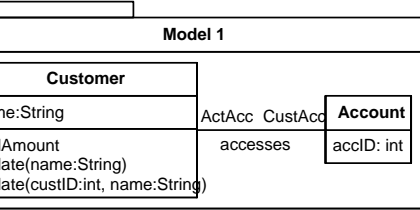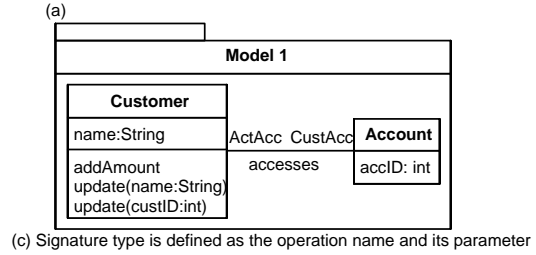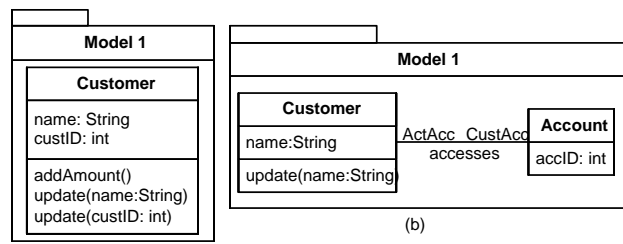If the model composition algorithm is applied on the pack-

Figure 5. A simple example illustrating composition variants

ages, the algorithm will compose the packages based on the signature type. If the signature type has been defined as model name, class name, and operation name with parameter name and type, the algorithm will produce the composed model shown in Figure 5(c). On the other hand if the signature type is defined as the model name, class name, and operation name without the parameter name and type, the composition will produce the model shown in Figure 5(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 the composition for the example can be done manually. In cases where there are a large number of classes and associations, this becomes extremely complex. The KerMeta implentation the algorithm can resolve the problem. 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 as the developer desires.

We have used the implementation to compose 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 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 rather than elements being merged. The composition in AOCE occurs at run-time and 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 using the ≪aspect≫ stereotype. Composition semantics follow the AspectJ implementation of AOP. Since 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. 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 that occur 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, and provide decision support 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 a signature-based mechanism rather than name-based mechanism. Composition of aspect models and a primary model may produce conflicts and undesirable emergent behavior, some of which can be detected by the composition technique.

We have developed tool using KerMeta to support the signature based composition technique. The inputs to the tool are an aspect model and a primary model. The output is a composed model that can be used as a primary model to compose with another aspect model. The composition procedure can detect conflicts but the resolution of conflicts still needs to be handled explicitly by the developer. Composition directives can be used to resolve conflicts. We are currently research the changes required to the algorithm to include composition directives.

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. *IEE 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] 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.

[13] 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.

[14] The Object Management Group. Unified Modeling Language: Superstructure. Version 2.0, OMG, ptc/03-07-06, 2003.

APPENDIX

A.   Merge part of the Composition algorithm

```
*******************************************************************************************
// e1 and e2 are the model elements that need to be merged
e1.merge(e2 : ModelElement)          //precondition : e1.sigEquals(e2) returns true
*******************************************************************************************
result := e1.getMetaClass.new // create the merged instance in the context of e1

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

  if type of p is primitive
  // Primitive type is the basic datatype 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 object has 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())
```