DSMDiff: A Differentiation Tool for Domain-Specific Models

Yuehua Lin[1], Jeff Gray[1], and Frédéric Jouault[2, 1]

[1]*Department of Computer and Information Sciences*
University of Alabama at Birmingham
Birmingham AL 35294-1170
{liny, gray, jouault} @ cis.uab.edu

[2]*ATLAS group (INRIA & LINA)*
University of Nantes, France

KEYWORDS

Model-Driven Engineering, Domain-Specific Modeling, Model Comparison

ABSTRACT

Model differentiation techniques, which provide the capability to identify mappings and differences between models, are essential to many model development and management practices. There has been initial research toward model differentiation applied to UML diagrams, but differentiation of domain-specific models has not been explored deeply in the modeling community. Traditional modeling practice using the UML relies on a single fixed general-purpose language (i.e., all UML diagrams conform to a single metamodel). In contrast, Domain-Specific Modeling (DSM) is an emerging model-driven paradigm in which multiple metamodels are used to define various modeling languages that represent the key concepts and abstractions for particular domains. Therefore, domain-specific models may conform to various metamodels, which requires model differentiation algorithms be metamodel-independent and able to apply to multiple domain-specific modeling languages. This paper presents metamodel-independent algorithms and associated tools for detecting mappings and differences between domain-specific models, with facilities for graphical visualization of the detected differences.

INTRODUCTION

Model-Driven Engineering (MDE) is emerging as a software development paradigm that promotes models as first-class artifacts to specify properties of software systems at a higher level of abstraction. The capability to identify mappings and differences between models, which is called model differentiation or model comparison, is essential to many model development and management practices [Cicchetti et al., 2007]. For example, model differentiation is needed in a model versioning system to trace the changes between different model versions to understand the evolution history of the models. Model comparison techniques and tools may help maintain consistency between different views of a modeled system. Furthermore, model differentiation can also be applied to assist in testing the correctness of model transformations by comparing the expected model and the resulting model after applying a transformation ruleset.

Although there exist many techniques available for differentiating text files (e.g., source code and documentation) and for structured data (e.g., XML documents), such tools either operate under a linear file-based paradigm that is purely textual (e.g., the Unix diff tool [Hunt and McIlroy, 75]) or perform comparison on a tree structure (e.g., the XMLDiff tool [Wang et al., 03]). However, models are structurally represented as graphs and are often rendered in a graphical notation. Thus, there is a structural mismatch between currently available text-based differentiation tools and the graphical nature of models. Furthermore, from our experience, large models can contain several thousand modeling elements, which makes a manual approach to model differentiation infeasible. To address these problems, more research is needed to explore automated differentiation algorithms and supporting tools that may be applied to models with graphical structures.

Theoretically, generic model comparison is similar to the graph isomorphism problem that can be defined as finding the correspondence between two given graphs, which is known to be NP-hard [Khuller and Raghavachari, 96]. Some research efforts aim to provide generic model comparison algorithms, such as the Bayesian approach, which initially provides diagram matching solutions to architectural models and data models [Mandelin et al., 06]. However, the computational complexity of general graph matching algorithms is the major hindrance to applying them to practical applications in modeling. Thus, it is necessary to loosen the constraints on graph matching to find solutions for model comparison. A typical solution is to provide differentiation techniques that are specific to a particular modeling language, where the syntax and semantics of this language help handle conflicts during model matching.

Currently, there exist many types of modeling languages. Particularly, the Unified Modeling Language (UML) is a popular object-oriented modeling language. The majority of investigations into model differentiation focus on UML diagrams [Ohst et al., 03], [Xing and Stroulia, 05]. Alternatively, Domain-Specific Modeling (DSM) [Gray et al., 07] is an emerging MDE methodology that generates customized modeling languages and environments from metamodels that define a narrow domain of interest. DSM has been adopted frequently in the development of computer-based systems, especially in the domain of embedded control software (e.g., avionics and automotive control).

Distinguished from UML, which is a general-purpose modeling language, Domain-Specific Modeling Languages (DSMLs) aim to specify the solution directly using rules and concepts familiar to end-users of a particular application domain. There are two main differences between domain-specific models and UML diagrams: 1) UML diagrams have a single definition for syntax and static semantics (i.e., a single metamodel); however, domain-specific models vary significantly in their structures and properties when their syntax and static semantics are defined in different metamodels, which correspond to different DSMLs tailored for specific end-users; 2) domain-specific models are usually considered as instance-based models (e.g., large domain-specific system models often have repetitive and nested hierarchical structures and may contain large quantities of objects of the same type), but traditional UML diagrams are primarily class-based models. Thus, domain-specific models and UML diagrams differ in structure, syntax and semantics. New approaches are therefore required to analyze differences among domain-specific models. However, there has been little work reported in the literature on computing differences between domain-specific models that are visualized in a graphical concrete syntax. The main goal of this paper is to present our algorithms that are metamodel-independent and have been implemented in a tool called DSMDiff, which addresses the problem of computing the differences between domain-specific models by exploring the following issues:

- Q1. What are the essential characteristics of domain-specific models and how are they defined?
- Q2. What information within domain-specific models needs to be compared and what information is needed to support metamodel-independent model comparison?
- Q3. How is this information formalized within the model representation in a particular DSML?
- Q4. How are model mappings and differences defined to enable model comparison?
- Q5. What algorithms can be used to discover the mappings and differences between models?
- Q6. How to visualize the result of model comparison to assist in comprehending the mappings and differences between two models?

The next section (*Metamodeling and Domain-Specific Models*) provides a foundation for discussing the key entities of domain-specific modeling that contribute to model comparison (i.e., questions Q1 through Q3). The section entitled *Model Differences and Mappings* offers a context for Q4, which addresses the mapping and difference sets used in model comparison. The core of the paper makes a contribution to model differentiation (see the section on *Model Differentiation Algorithms*, which addresses Q5) by describing the algorithms that we have developed and implemented for a model differentiation tool. This section also motivates the importance of visualizing the model differences in a manner that can be comprehended by a model engineer, which is the essence of Q6. The paper presents an evaluation of the algorithms (*Evaluation and Discussion*) and concludes with an overview of related work and a summary conclusion.

METAMODELING AND DOMAIN-SPECIFIC MODELS

To develop algorithms for model differentiation, one of the critical questions is whether to determine if the two models are syntactically equivalent or to determine if they are semantically equivalent. Because the semantics of most modeling languages are not formally defined, the algorithms presented in this paper only determine whether the two models are syntactically equivalent[1]. To achieve this, a model comparison algorithm must be informed by the syntax of a specific DSML. Thus, this section discusses how the syntax of a DSML is defined and what essential information is embodied in the syntax.

Metamodeling is a common technique for conceptualizing a domain by defining the abstract syntax and static semantics of a DSML. A metamodel defines a set of modeling elements and their valid relationships that represent certain properties for a specific domain. The Generic Modeling Environment (GME) [Lédeczi et al., 01] is a meta-configurable tool that allows a DSML to be defined from a metamodel. Domain-specific models can be created using this DSML and may be translated into source code, or synthesized into data to be sent to a simulation tool. The work described in this paper was performed within the context of the GME, but we believe the algorithms that are described can solve broader model comparison problems in other metamodeling tools that represent models as hierarchical graphs, such as the ATLAS Model Management Architecture (AMMA) [Kurtev et al., 06], Microsoft's DSL tools [Microsoft, 05], MetaEdit+ [MetaCase, 07], and the Eclipse Modeling Framework (EMF) [Budinsky et al., 04].

There are three basic types of entities used to define a DSML in GME: *atom*, *model* and *connection*. An *atom* is the most basic type of entity that cannot have any internal structures. A *model* is another type of entity that can contain other modeling entities such as child models and atoms. A *connection* represents the relationship between two entities. Generally, the constructs of a DSML defined in a metamodel consist of a set of model entities, a set of atom entities and a set

[1] Please note that this is not a serious limitation when compared to other differentiation methods. The large majority of differentiation techniques offer syntactic comparison only, especially those focused on detecting textual differences.

of connections. However, these three types of entities are generic to any DSML and provide domain-independent type information (i.e., called the *type* in GME terminology). Each entity (e.g., model, atom or connection) in a metamodel is given a name to specify the role that it plays in the domain. Correspondingly, the name that is defined for each entity in a metamodel represents the domain-specific type (i.e., called the *kind* in GME terminology), which end-users see when creating an instance model. Moreover, *attributes* are used to record state information and are bound to atoms, models, and connections. Thus, without considering its relationships to other elements, a model element is defined syntactically by its *type*, *kind*, *name* and a set of *attributes*. Specifically, *type* provides certain meta information to help determine the essential structure of a model element for any DSML (e.g., model, atom or connection) and is needed in metamodel-independent model differentiation algorithms. Meanwhile, *kind* and *name* are specific to a given DSML and provide non-structural syntactical information to further assist in model comparison. Other syntactical information of a model element includes its relationships to other elements (i.e., *connections* to its neighbours), which may also distinguish the identity of modeling elements.

In summary, to determine whether two models are syntactically equivalent, model differentiation algorithms need to compare all the syntactical information between them. Such a set of syntactical information of a model element include: 1) its *type, kind, name* and *attribute* information; and 2) its connections to other model elements. There is other information associated with a model that either relates to the concrete syntax of a DSML (e.g., visualization specifications such as associated icon objects and their default layouts and positions) or to the static semantics of a DSML (e.g., constraints to define domain rules). The concrete syntax is not generally involved in model differentiation for the purpose of determining whether two models are syntactically equivalent (e.g., the associated icon of a model element is always determined by its *kind* information by the metamodel definition). Similarly, because the constraints are defined at the metamodel level in our case (i.e., models with the same *kind* hold the same constraints), they are not explicitly computed in model differentiation; instead, *kind* equivalence implies the equivalence of constraints.

Graph Representation of Domain-Specific Models

In order to design efficient algorithms to detect differences between two models, it is necessary to understand the structure of a model. Figure 1 shows a GME model and its hierarchical structure. According to its hierarchical containment structure, a model can be represented formally as a hierarchical graph that consists of a set of nodes and edges, which are typed, named and attributed. There are four kinds of elements in such a graph:

- **Node.** A node is an element of a model, represented as a 4-tuple (*name*, *type*, *kind*, *attributes*), where *name* is the identifier of the node, *type* is the corresponding metamodeling element for the node, *kind* is the domain-specific type, and *attributes* is a set of attributes that are predefined by the metamodel. There are two kinds of nodes:
    - **Model node:** a containment node that can be expanded at a lower level as a graph that consists of a set of nodes and a set of edges (i.e., a container). This kind of node is used to represent submodels within a model, which leads to multiple-level hierarchies of a containment model.
    - **Atom node:** an atomic node that cannot contain any other nodes (i.e., a leaf). This kind of node is used to represent atomic elements of a model.
- **Edge.** An edge is a 5-tuple (*name*, *type*, *kind*, *src*, *dst*), where *name* is the identifier of the edge, *type* is the corresponding metamodeling element for the edge, *kind* is the domain-specific type, *src* is the source node, and *dst* is the destination node. A connection can be represented as an edge.
- **Graph.** A directed graph consists of a set of nodes and a set of edges where the source node and the destination node of each edge belong to the set of nodes. A graph is used to represent an expanded model node.

- **Root.** A root is the graph at the top level of a multiple-level hierarchy that represents a root model.

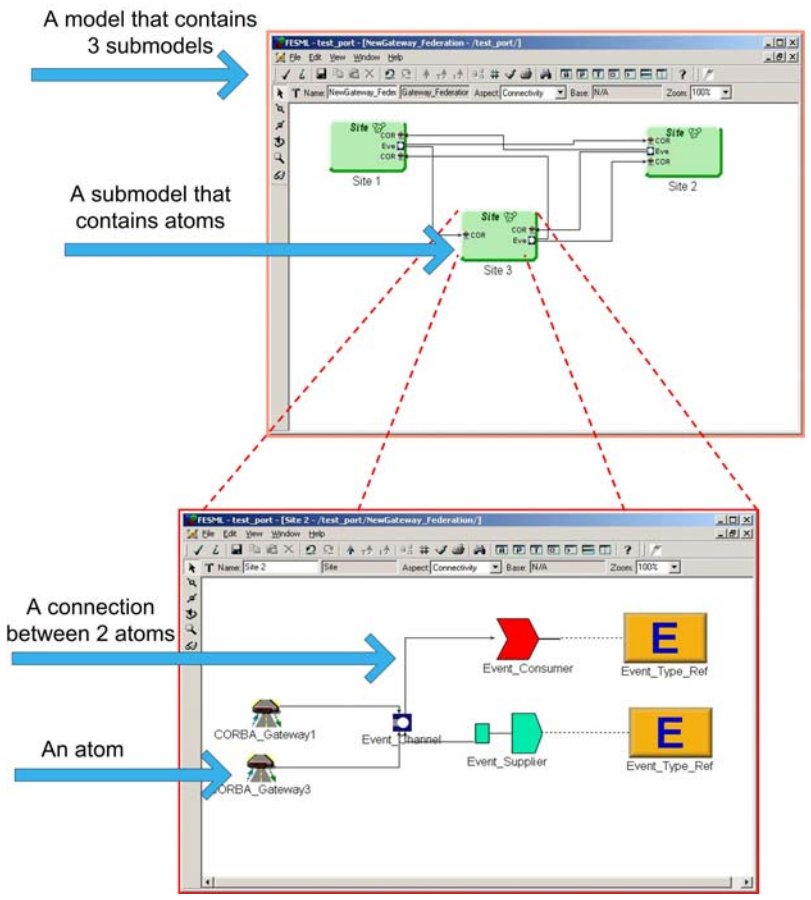


Figure 1. A GME model and its hierarchical structure

MODEL DIFFERENCES AND MAPPINGS

The task of model differentiation is to identify the mappings and differences between two containment models at all hierarchical levels. In general, the comparison starts from the top levels of the two containment models and then continues to the child submodels. At each level, the comparison between two corresponding models (i.e., one is defined as the host model, denoted as M1, and the other is defined as the candidate model, denoted as M2), always produces two sets: the mapping set (denoted as MS) and the difference set (denoted as DS). The mapping set contains all pairs of model elements that are mapped to each other between two models. The difference set contains all detected discrepancies between the two models. Before the details of the algorithms are presented, the definition of model mappings and differences is discussed.

A pair of mappings is denoted as **Map** ($elem^1$, $elem^2$), where $elem^1$ is in M1 and $elem^2$ is in M2, and may be a pair of nodes or a pair of edges. **Map** ($elem^1$, $elem^2$) is a bidirectional relationship that implies $elem^2$ is the only mapped correspondence in M2 for $elem^1$ in M1 based on certain matching metrics, and vice versa. The difference relationship between two models is more complicated than the mapping relationship. The notations used to represent the differences between two models are editing operational terms that are considered more intuitive [Alanen and Porres, 03]. For example, a **New** operation implies creating a model element, a **Delete** operation implies removing a model element and a **Change** operation implies changing the value of an attribute. We define DS = M2 – M1, where M2 is compared to M1. DS consists of a set of operations that yields M2 when applied to M1. The "-" operator is not commutative.

There are several situations that could cause two models to differ. The first situation of model difference occurs when some modeling elements (e.g., nodes or edges in the graph representation) are in M2, but not in M1. We denote this kind of difference as **New** (e^2) where e^2 is in M2, but not in M1. The converse is another situation that could cause a difference (i.e., elements in M1 are missing in M2). We denote this kind of difference as **Delete** (e^1) where e^1 is in M1, but not in M2. These two situations occur from structural differences between the two models. A third difference can occur when all of the structural elements are the same, but a particular value of an attribute is different. We denote this difference as **Change** (e^1, e^2, f, v^1, v^2), where e^1 in M1 and e^2 in M2 are a pair of mapping elements, f is the feature name (e.g., name of an attribute), v^1 is the value of e^1.f, and v^2 is the value of e^2.f. Thus, the difference set actually includes three sets: DS = {N, D, C} where N is a set that contains all the **New** differences, D is a set that contains all the **Delete** differences, and C is a set that contains all the **Change** differences. These definitions were initially presented in [Lin et al., 05].

MODEL DIFFERENTIATION ALGORITHMS

Our model comparison algorithms identify the mappings and differences between two containment models by comparing all the elements and their abstract syntactical information within these models. In general, the comparison starts from the two root models and then continues to the child submodels. At each level, two metrics (i.e., signature matching and structural similarity) are combined to detect the mapped nodes between a pair of models and the remaining nodes are examined to determine all the node differences. Based on the results of node comparison, all the edges are computed to discover all the edge mappings and differences.

To store the two models that need to be compared and the results of model comparison, a data structure called DiffModel is used. The structure of DiffModel contains a pair of models to be compared, a mapping set to store all the matched child pairs, and three difference sets to record **New**, **Delete,** and **Change** differences.

Detection of Model Mappings

It is well-known that some model comparison algorithms are greatly simplified by requiring that each element have a persistent identifier, such as a universally unique identifier (UUID), which is assigned to a newly created element and will not be changed unless the element is removed [Ohst et al., 03]. However, such traceable links only apply to two models that are subsequent versions. In many modeling activities, model comparison is needed between two models that are not subsequent versions. A pair of corresponding model elements need to share a set of properties, which can be a subset of their syntactical information. Such properties may include type information, which can be used to select the model elements of the same type from the candidates to be matched because only model elements with the same type need to be compared. For example, in a Petri net model, a “place” node will not match a “transition” node. In addition to type information, identification information such as name is also important to determine mappings for domain-specific models. Therefore, a combination of syntactical properties for a node or an edge can be used to identify different model elements. Such properties are called the *signature* in DSMDiff, and are used as the first criterion to match model elements. Signature is a widely used term in much of the literature on structural data matching and may have different definitions [Wang et al., 03]. In our context, the signature of a node or an edge is a subset of its syntactical information, which is defined as follows:

- **Node Signature** is the concatenation of the type, kind and name of a node. Suppose *v* is a node in a graph. *Signature (v) = /Type (v)/Kind (v)/Name (v)*. If a node is nameless, its name is set as an empty string.

- **Edge Signature** is the concatenation of the type, kind and name of the edge as well as of the signatures of its source node and destination node. Suppose *e* is an edge in a graph, *src* is its source node and *dst* is its destination node. *Signature (e) = Signature (src)/Type (e)/Kind (e)/Name (e)/Signature (dst)*. If an edge is nameless, its name is set as an empty string.

Signature Matching

Signature matching can be defined as:

- **Node Signature Matching:** Given two models, M1 and M2, suppose v^1 is a node in M1 and v^2 is a node in M2. There is a node signature matching between v^1 and v^2 if Signature (v^1) = Signature (v^2), which means the two strings (i.e., the signature of v^1 and the signature of v^2) are textually equivalent.

- **Edge Signature Matching:** Given two models, M1 and M2, suppose e^1 is an edge in M1 and e^2 is an edge in M2. There is an edge signature matching between e^1 and e^2 if *Signature (e^1) = Signature (e^2)*, which means the two strings (i.e., the signature of e^1 and the signature of e^2) are textually equivalent.

A node v^1 in M1 mapping to a node v^2 in M2 implies their name, type and kind are matched. An edge e^1 in M1 mapping to an edge e^2 in M2 implies their name, type, kind, source node and destination node are all matched.

Usually, nodes are the most significant elements in a model and edge mappings also depend on whether their source and destination nodes match. Thus, DSMDiff first tries to match nodes that have the same signature. For example, to decide whether there is a node in M2 mapped to a node in M1 (denoted as v^1), the algorithm first needs to find all the candidate nodes in M2 that have the same signature as v^1 in M1. If there is only one candidate found in M2, the identified candidate is considered as a unique mapping for v^1 and they are considered as syntactically equivalent. If there is more than one candidate that has been found, the signature cannot identify a node uniquely. Therefore, v^1 and its candidates in M2 will be sent for further analysis where structural matching is performed.

Structural Matching

In some cases, signature matching alone cannot find the exact mapping for a given model element. During signature matching, one node in M1 may have multiple candidates in M2. To find a unique mapping from these candidates, DSMDiff uses structural similarity as another criterion. The metric used for determining structural similarity between a node and its candidates is called edge similarity, which is defined as follows:

Edge Similarity: Given two models, M1 and M2, suppose v^1 is a node in M1 and v^2 is one of its candidate nodes in M2. The edge similarity of v^2 to v^1 is the number of edges connecting to v^2, with each signature matched to one of the edges connecting to v^1.

During structural matching, if DSMDiff can find a candidate that has the *maximal edge similarity*, this candidate becomes the unique mapping for the given node. If it cannot find this unique mapping using edge similarity, one of the candidates will be selected as the host node's mapping, following the assumption that there may exist a set of identical model elements.

Listing 1 presents the algorithm to find the candidate node with maximal edge similarity for a given host node from a set of candidate nodes. It takes the host node (i.e., `hostNode`) and a set of candidate nodes of M2 (i.e., `candidateNodes`) as input, computes the edge similarity of every candidate node and returns a candidate with maximal edge similarity. Listing 2 gives the algorithm for computing edge similarity between a candidate node and a host node. It takes two maps as input – `hostConns` stores all the incoming and outgoing edges of the host node indexed by their edge signature, and `candConns` stores all the incoming and outgoing edges of the candidate node indexed by their edge signature. By examining the mapped edge pairs between these two maps, the algorithm computes the edge similarity as output.

The algorithm in Listing 1 determines that the unique correspondence found using edge similarity has the most identical connections and neighbors to the host node. The algorithm also implies one candidate with the maximal edge similarity is selected as the unique correspondence when there is more than one candidate with the same maximal edge similarity; however, it may be incorrect in some cases and needs to be improved as discussed later in the *Limitations and Improvement* section. DSMDiff only examines structural similarity within a specific local region where the host node is the center and its neighbor nodes form the border. In our experience, using

```
Name: findMaximalEdgeSimilarity
Input: hostNode, candidateNodes
Output: maximalCandidate

   1. Initialize three maps: hostConns, candConns and set
      maxSimilarity = 0, maximalCandidate = null;
   2. Store each edge signature and the number of associated
      edges of the hostNode in the map hostConns;
   3. For each candidate c in candidateNodes
          1) Store each of its edge signatures and the number of
             associated edges in the map candConns;
          2) Call computeEdgeSimilarity(hostConns, candConns)to
             compute the edge similarity of c to hostNode;
          3) if( the computed similarity > maxSimilarity)
                 maxSimilarity = the computed similarity;
                 maxmalCandidate = c;
   4. Return maximalCandidate;
```

Listing 1. Finding the candidate of maximal edge similarity

```
Name: computeEdgeSimilarity
Input: hostConns, candConns
Output: similarity

   1. Initialize similarity as zero;
   2. For each edge signature in the map hostConns
         1) Get the number of the edges associated with the
            edge signature as hostCount;
         2) Get the number of the edges from the map candConns
            associated with the edge signature as candCount;
         3) If candCount <= hostCount
              Similarity = similarity + candCount;
         4) Else
              Similarity = similarity + hostCount;
   3. Return similarity;
```

Listing 2. Computing edge similarity of a candidate

signature matching and edge similarity to find model mappings not only speeds up the model differentiation process, but also generates results with acceptable accuracy in general practice. After all the nodes in M1 have been examined by signature and structural matching, all the possible node mappings between M1 and M2 are found.

Determination of Model Differences

As mentioned previously, there are three basic types of model differences: **New**, **Delete** and **Change**. To identify these various types of differences is another major task of DSMDiff. In order to increase the performance of DSMDiff, some of the processes to detect model differences may be integrated into the previously discussed processes to find mappings.

To discover all the **Delete** differences, DSMDiff must find all the model elements in M1 that do not have any signature matched candidates in M2. In signature matching, DSMDiff examines how many candidates can be found in M2 that have the same signature as each element in M1. If only one is found, a pair of mappings is constructed and added to the mapping set. If more than one is found, the host element and the found candidates are sent to structural matching. If no candidate can be identified, the host element is considered as a **Delete** difference, which means it exists in M1 but does not exist in M2. Listing 3 summarizes the algorithm.

```
Name: findSignatureMappingsAndDeleteDiffs
Input: diffModel
Output: hostSet, candMap, diffModel

     1. Initialize a set hostSet and a map candMap;
     2. Get M1 from diffModel and store all nodes of M1 in
        hostSet
     3. Get M2 from diffModel and store all nodes of M2 in
        candMap associated with their signature;
     4. For each node e1 in hostSet
          1) Get the count of the nodes from candMap that are
             signature matched to e1;
          2) If count == 1
               Get the candidate from candMap as e2;
               Add Map(e1, e2) to the mapping set of diffModel;
               Erase e1 from hostSet;
               Erase e2 from candMap;
          3) If count == 0
               Add e1 to the Delete set of diffModel;
               Erase e1 from hostSet;
          4) If count > 1
               Do nothing;
```

Listing 3. Finding signature mappings and the Delete differences

After all the mappings are discovered between M1 and M2, the mapped elements are filtered out. The remaining elements in M2 are then taken as the **New** differences (i.e., a **New** difference indicates that there is an element in M2 that is missing in M1).

The **Change** differences are used to indicate varying attributes between any pair of mappings. Both model nodes and atom nodes may have a set of attributes, thus a pair of matched model nodes or atom nodes may have **Change** differences. DSMDiff compares the values of each attribute of each pair of model or atom mappings. If the values are different, the attribute name is added to the **Change** difference set.

After all the node mappings and differences are determined, DSMDiff then tries to find the edge mappings and differences between M1 and M2 using these strategies: 1) all the edges connecting to a **Delete** node are **Delete** edges; 2) all the edges connecting to a **New** node are **New** edges; 3) the edge signature matching is applied to find out the edge mappings; and 4) the remaining edges in M1 are taken as additional **Delete** edges and those in M2 are taken as additional **New** edges.

Depth-first Detection

The traversal strategy of DSMDiff is depth-first, which traverses from the root level of a model hierarchy and then walks down to the lower levels to compare all the child submodels until it reaches the bottom level, where there are no submodels that can be expanded. Supporting such depth-first detection requires that all the node mappings found at a current level be categorized into two groups: model node mappings and atom node mappings. DSMDiff then performs model comparison on each pair of model node mappings. Each atom node mapping is examined for attribute equivalence. If there are some attributes with different values, these represent **Change** differences between the models. If all the attributes are matched, it is inferred that two nodes are equivalent because there is no **Change**, **Delete** or **New** difference.

To summarize, Listing 4 presents the overall algorithm of DSMDiff to calculate the mappings and the differences between two models. It takes `diffModel` as input, which is a typed DiffModel and initially stores two models (M1 and M2). DSMDiff produces two sets: the mapping set (MS) and the difference set (DS) that consists of three types of differences (N: the set of New differences, D: the set of Delete differences, and C: the set of Change differences). All of these mapping and difference sets are stored in the `diffModel` during execution of DSMDiff.

```
Name: DSMDiff
Input: diffModel
Output: diffModel

   1. Initialize a set hostSet and a map candMap;
   2. Get the host model from diffModel as M1 and the
      candidate model as M2;
   3. Detect attribute differences between M1 and M2 and add
      them to the Change set of diffModel;
   4. //Find node mappings by signature matching
      findSignatureMappingsAndDeleteDiffs (diffModel,
                                           hostSet, candMap);
   5. If(hostSet is not empty && candMap is not empty)
            //Find node mappings by structural matching
            For each element e1 in hostSet
                 1) Get its candidates from candMap into a set
                    called candSet;
                 2) e2 = findMaximalEdgeSimilarity(e1,candSet);
                 3) Add Pair(e1, e2) to the Mapping set of
                    diffModel;
                 4) Erase e1 from hostSet;
                 5) Erase e2 from candMap;
   6. If(candMap is not empty)
            Add all the remained members of candMap to the New
            set of diffModel;
   7. For each mapped elements that are not submodels
            Detect attribute differences and add them to the
            Change set of diffModel;
   8. Compute edge mappings and differences
   9. //Walk into child submodels
      For each childDiffModel that stores a pair mapped
      submodels
            DSMDiff(childDiffModel);
```

Listing 4. DSMDiff Algorithm

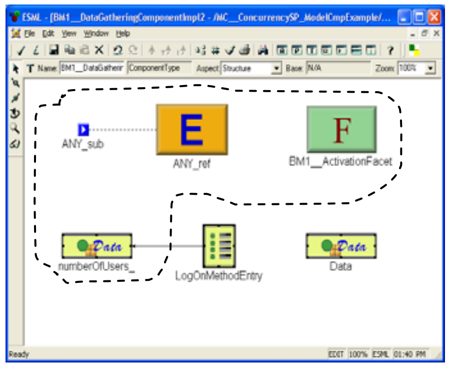
Visualization of Model Differences

Visualization of the result of model differentiation (i.e., structural model differences) is critical to assist in comprehending the mappings and differences between two models. To help communicate the comparison results intuitively within a host modeling environment, a tree browser has been developed to visualize the structural differences and to support navigation among the analyzed model differences.

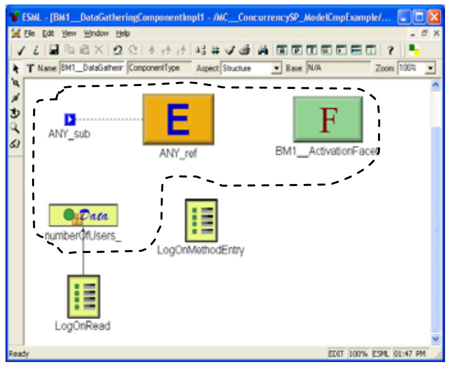
This browser looks similar to the model browser of GME, using the same graphical icons to represent items with types of model, atom and connection. To indicate the various types of differences, the browser uses three colors: red for a **Delete** difference, gray for a **New** difference, and green for a **Change** difference. The model difference browser displays two symmetric difference sets in two containment change trees: one indicates the difference set DS = M2 – M1 by annotating M1 with colors; and the other indicates the difference set DS' = M1 - M2 = -DS by annotating M2 with colors. If DS = {**New** = N, **Delete** = D, **Change** = C} then DS' = {**New** = D, **Delete** = N, **Change** = C}. For example, if there is a **Delete** difference in M1, correspondingly there is a **New** difference in M2. Such a symmetric visualization helps comprehend the corresponding relationships between two hierarchical models.

Figure 2 shows screenshots of two models and the detected differences in the model difference browser[2]. The host model M1 is shown in Figure 2-a, and the candidate model M2 is shown in Figure 2-b. The corresponding model elements within the encircled regions in these two models are the mappings, which are filtered and not displayed in the browser. The browser only visualizes the detected differences, as shown in Figure 2-c. The root of the upper tree is M1, its subtrees and leaf nodes are all the differences compared to M2, which is represented by the bottom tree. For example, the first child of the upper tree is a **Delete** difference, which is in red. This difference means the `LogOnRead` element is in M1, but is missing in M2. Correspondingly, there is a **New** difference in the bottom tree, which is in gray. It indicates that M2 misses the `LogOnRead` element when it is compared to M1. A **Change** difference is detected for the `LogOnMethodEntry` element; although this element exists in both models, one of its attributes, called *kind,* has different values: "On Write" in M1 but "On Method Entry" in M2. Such a **Change** difference item is in green. When the two trees do not have any subtree or leaf node, we can infer there is no difference between these two models. To focus on any model element, a user can navigate across the tree and double-click an item of interest, and the corresponding model element is brought into focus within the editing window.

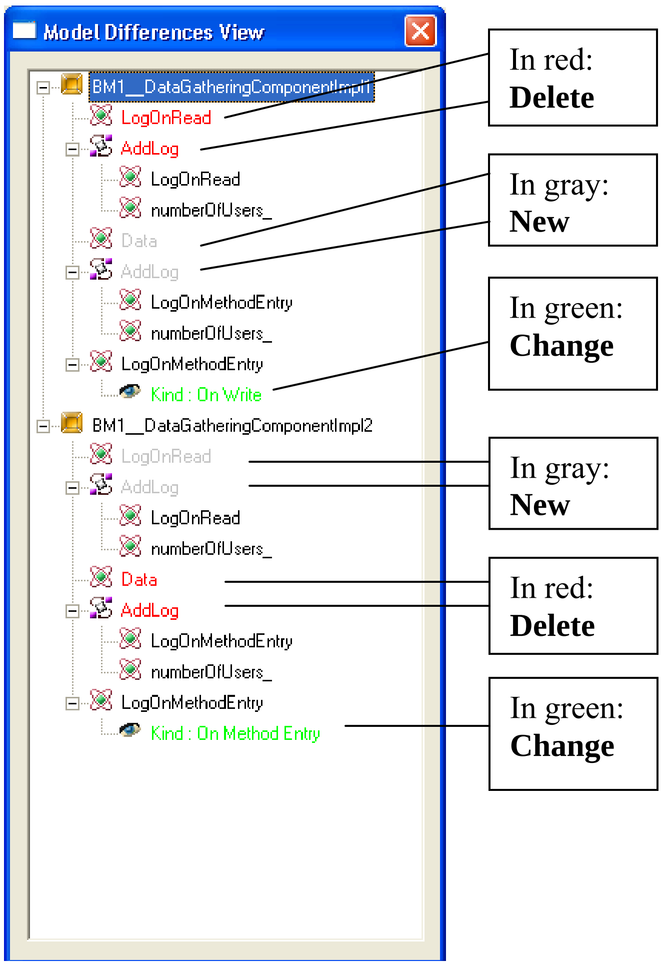
[2] Because the actual color shown in the browser can not be rendered in print, the figure has annotations that indicate the appropriate color.



(a) The host model: M1



(b) The candidate model: M2



```
BM1__DataGatheringComponentImpl1
    LogOnRead
    AddLog
        LogOnRead
        numberOfUsers_
    Data
    AddLog
        LogOnMethodEntry
        numberOfUsers_
    LogOnMethodEntry
        Kind : On Write
BM1__DataGatheringComponentImpl2
    LogOnRead
    AddLog
        LogOnRead
        numberOfUsers_
    Data
    AddLog
        LogOnMethodEntry
        numberOfUsers_
    LogOnMethodEntry
        Kind : On Method Entry
```

(c) Model differences

Figure 2. Visualization of model differences

EVALUATION AND DISCUSSION

This section first briefly analyzes the complexity of the algorithm and illustrates an example application. The current limitations and proposed improvements for DSMDiff are also discussed.

Algorithm Analysis

Generally, DSMDiff is a level-wise model differentiation approach. It begins with the two root models at the top levels and then continues to their child models at the lower levels. At each level, node comparison is performed to detect the node mappings by using signature matching and edge similarity, followed by edge comparison to detect the edge mappings and differences. These steps are repeated on the mapped child models until the bottom level is reached.

The core of the DSMDiff algorithms includes signature matching (Step 4 in Listing 4) and edge similarity matching (Step 5 in Listing 4), which significantly influence the execution time. To estimate the complexity of signature matching and edge similarity matching, we assume the two models have similar structures and sizes. Given a model, L denotes the depth of the model hierarchy; N denotes the average number of nodes; and, M denotes the average number of the edges of a model node. The size of a model node is denoted as S, where S = N + M. Considering the case that every node at all levels except for the lowest level are model nodes, the total number of model nodes is denoted as T, where T = $\sum_{i=0}^{L-2} N^{i} \approx N^{L-1}$.

In the best case, all the mappings and differences between two model nodes can be found by signature matching, in which the complexity depends on the size of the model nodes. In `findSignatureMappingsAndDeleteDiffs` (Listing 3), where signature matching is performed to detect node mappings and differences, all the candidate nodes and their signatures are stored in a sorted map; the upper bound for the complexity of this step is O(N x logN). To find correspondences from this map for all the node elements of M1, the complexity is also O(N x logN). Later, similar computation is taken to compute the edge mappings and differences (i.e., Step 8 of Listing 4); such complexity is neglected here because the number of edges is less than the number of nodes. Overall, because all the model nodes within the model hierarchy need to be compared, the complexity for this best case is O(N x logN x T).

In the worst case, no exact mapping is found for a pair of model nodes during the signature matching. Thus, all the nodes need to be examined by edge similarity matching (i.e., Step 5 in Listing 4), which is the most complicated step in Listing 4. Assume that there is an edge between any pair of nodes, then a node has N-1 edges, which is the worst case regarding the complexity. In edge similarity matching, the most complicated step is `findMaximalEdgeSimilarity`

(Listing 1) that computes the edge similarity of all the candidate nodes for a host node, where all edge signatures of each candidate node and the number of the associated edges are stored in a map (i.e., Substep 3.1 in Listing 1). The complexity for building this map is O({N-1} x log{N-1}). To compute the edge similarity of every candidate node (i.e., Step 3 of Listing 1), the computing cost is bound by O(R x {N-1} x log{N-1}), where R is the number of candidate nodes with R ≤ N. Because Step 3 is the most complicated step in Listing 1, the upper bound of `findMaximalEdgeSimilarity` is also O(R x {N-1} x log{N-1}). To find the candidate with maximal edge similarity for each host node (i.e., Step 5 in Listing 4), the cost is bounded by O(N x R x {N-1} x log{N-1}). To compute all the node mappings at all the levels in a model hierarchy using edge similar matching, the upper bound of the complexity for this worst case is O(T x N x R x {N-1} x log{N-1}), which is in the polynomial class. For the same reason (i.e., the number of edges is less than the number of nodes), the complexity of detecting edge mappings and differences is neglected.

Although the complexity of constant-time signature comparison and associated string comparison is not counted here, the algorithm achieves polynomial time in complexity according to the above analysis.

Application Example

Model-to-model transformation is one of the core activities to facilitate change evolution within MDE [Sendall and Kozaczynski, 03]. To ensure the correctness of model transformation, executable testing can help detect errors in a model transformation specification. The development of DSMDiff was originally motivated by research on model transformation testing [Lin et al., 05].

Inside the framework of model transformation testing, a testing engine is constructed to support execution of a finite set of test cases against a specific model and associated transformations. There are two components inside such a testing engine: one is the executor, the other is the model comparator. The executor is responsible for executing the to-be-tested transformation specifications. The functionality of the model comparator includes comparison of the actual output model and the expected model, and visualization of the test results. If there is no difference between the actual output and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there are differences between the output and expected models, the errors in the transformation specification need to be isolated and removed.

To realize the vision of model transformation testing, differentiation is performed between an expected model and an output model that are not subsequent versions. The potential traceable links between two subsequent model versions, which are used in most differentiation algorithms to reduce the computation complexity, are not available between an expected model specified by a test engineer and an output model produced by a transformation tool. In this application, DSMDiff serves as a model comparator to perform the model comparison and visualize the produced differences.

Limitations and Improvement

DSMDiff is based on the assumption that domain-specific models are defined precisely and unambiguously. That is, domain-specific models are instances of a metamodel that can be distinguished from each other by comparing a set of properties of the elements and the connections to their neighbors. However, when there are several candidates with the same maximal edge similarity, DSMDiff may produce inaccurate results. A typical case occurs when there are nodes connected to each other but their mappings have not been determined yet. As shown in Figure 3, there is an A node connected to three nodes: B, C and D. In M2, the A' node connects to three other nodes (B', C' and D') and A'' is connected to B''. Given that nodes with the same letter label have the same signatures (e.g., all the A nodes have the same signature and all the B nodes have the same signature), then the connections between an A node and a B node have the same edge signature. According to the algorithm in Listing 1, suppose the A node is examined first for structural matching and the A’ node in M2 is selected as the mapping of the A node in M1. When the B node is examined, the algorithm may select either B’ or B’’ in M2 as the mapping of the B node in M1 because both B nodes in M2 have the same edge similarity as the B node in M1. If the B’’ node in M2 is selected as the mapping to the B node in M1, the result is incorrect because B’ is the correct mapping. In such cases, DSMDiff needs to use new rules or criteria to help find the correct mapping. For example, a new rule needs to be added to the algorithm in Listing 1 to require selecting first the unmapped node in M1 that has maximal already-mapped neighbors. Another improvement will allow interaction between DSMDiff and users, who can select the mappings from multiple candidates manually.

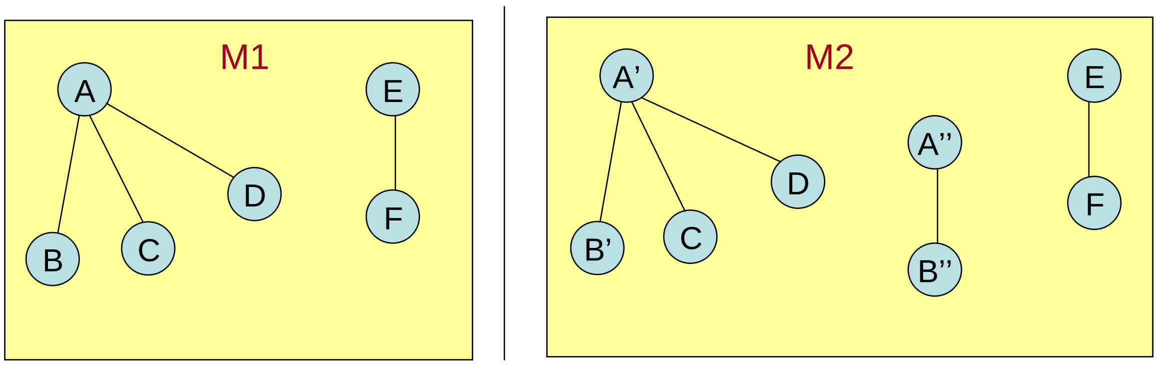


Figure 3. A nondeterministic case that DSMDiff may produce incorrect result

Besides the performance and the correctness of the results, it is also important for model differentiation algorithms to produce a small set of model differences (ideally a minimal set) rather than providing a large set of model differences. In other words, the conciseness of the produced result is another metric contributing to the overall quality of model differentiation algorithms. Currently, DSMDiff compares two models M1 and M2 by traversing their composition trees in parallel. When an element from a model cannot be matched to an element of the other model at some level, the algorithm does not traverse the children of this element. One issue with this scheme is that DSMDiff is not able to detect when a subtree has been moved from one container to another between M1 and M2. The algorithm will only report that a whole subtree has been deleted from M1, and that a whole subtree has been added to M2, without noting that these are identical subtrees. This implies that the reported difference set is less concise than it could be. To solve this problem, a new type of model difference needs to be introduced: **Move**, which may reference the subtree in M1, and its new container in M2. An additional step is also required in the algorithms to compare all the elements of M1 and M2 that have not been matched when first traversing them. However, this step is expensive in the general case because many elements may need to be compared. This cost is actually avoided in the current version of the algorithm by assuming a similar composition structure in M1 and M2.

DSMDiff visualizes all the possible differences as a containment tree in a browser, but does not directly highlight the differences upon the associated model elements within the editing window. To indicate the differences directly on the model diagrams and attribute panels within the modeling environment, a set of graphical decorators, which may be shapes or icons, could be attached to the corresponding model elements or attributes in order to change their look according to the type of model differences. In addition, our solution using coloring to highlight all possible types of model differences may fail to work when users are color-blind, or when a screenshot of the model difference tree view is printed in black-and-white (e.g., the need to add annotations to Figure 2-c). A visualization mechanism to complement the coloring would indicate the **Delete** differences by striking through them, the **Change** ones by underlining them, and marking the **New** ones bold. This could be a more efficient solution that needs to be investigated in the future.

RELATED WORK

This work is related to differentiation techniques for various software artifacts such as source code, documents, diagrams and models. There are two important categories of related work: 1) the algorithms to compute model differences, and 2) the visualization techniques to highlight those differences.

Model Differentiation Algorithms

There exist a number of general-purpose differentiation tools for comparing two or more text files (e.g., code or documentation). As an example, Unix diff [Hunt and McIlroy, 75] is a popular tool for comparing two text files. Diff compares files and indicates a set of additions and deletions. Many version control tools also provide functionality similar to diff to identify changes between versions of text documents [Eick et al., 01].

Although many tools are available for differentiating text documents, limited support is currently available for differentiating graphical objects such as UML diagrams and domain-specific models. As the importance of model differentiation techniques to system design and its evolution is well-recognized, there have been some research efforts focused on model difference calculation.

Several metamodel-independent algorithms regarding difference calculation between models are presented in [Alanen and Porres, 03] and [Ohst et al., 03], which are developed primarily based on existing algorithms for detecting changes in structured data [Chawathe et al., 96] or XML documents [Wang et al., 03]. In these approaches, a set of change operations such as "create" and "delete" are used to represent and calculate model differences, which is similar to our approach. However, they are based on the assumption that the model versions are manipulated through the editing tool that assigns persistent identifiers to all model elements. Such capability is not available when two models are developed separately (e.g., by different developers in a non-collaborative context, or by different editing tools) or generated by execution of a model transformation.

To provide algorithms independent of such identifiers, UMLDiff uses name similarity and structure similarity for detecting structural changes between the designs of subsequent versions of

UML models [Xing and Stroulia, 05]. However, differentiation applied to domain-specific modeling is more challenging than difference analysis on UML diagrams. The main reason is that UML diagrams usually belong to a single common metamodel that can be represented formally as a containment-spanning tree starting at a virtual root and progressing down to packages, classes and interfaces. However, domain-specific models may belong to different metamodels according to their domains and are considered as hierarchical graphs. Also, a differentiation algorithm for domain-specific models needs to be metamodel-independent in order to work with multiple DSMLs. This required DSMDiff to consider the type information of instance models, as well as the type information of the corresponding metamodel.

Visualization of Model Differences

There has been some work toward visualizing model differences textually. IBM Rational Rose [Rose, 07] and Magic Draw UML [MagicDraw, 07] display model differences in a textual way. These tools convert the diagrams into hierarchical text and then perform differentiation on this hierarchy. Changes are shown using highlighting schemes on the text. Although this approach is relatively easy to implement, its main drawback is that changes are no longer visible in a graphical form within the actual modeling tool, which makes the difference results more difficult to comprehend.

Other researchers have shown that the use of color and graphical symbols (e.g., icons) are more efficient in highlighting model differences. An approach is proposed in [Ohst et al., 03] where coloring is used to highlight the model differences in two overlapping diagrams. A differentiation tool described in [Mehra et al., 05] presents graphical changes by developing a core set of highlighting schemes and an API for depicting changes in a visual diagram. UMLDiff presents a change-tree visualization technique. It reuses the visualization of Eclipse's Java DOM model for displaying different entities with diverse icons and separate visibility with various colors. Additionally, UMLDiff extends the visualization to use different icons to represent the differentiation results (e.g., “+” for add, “-” for remove).

DSMDiff provides a model difference browser that displays the structural differences in a tree view, which is similar to the change-tree visualization technique of UMLDiff. To preserve the convention of the host modeling environment, many GME icons are used to represent the corresponding modeling types of the model difference items in the tree view. For example, a **Delete** atom or a **New** atom corresponds to an *atom* type. To avoid overuse of icons (e.g., “+” and “-” are commonly used for a collapsed folder and an expanded folder, respectively), DSMDiff uses colors to represent various types of model differences.

Although it is intuitive to visualize model differences by coloring and iconic notations, these techniques are not specifically tied to modeling concepts and lack the ability to be integrated into MDE processes. To address this problem, a promising approach is to represent the result of model difference as a model itself. A recent work presented in [Cicchetti et al., 07] proposes a metamodel-independent approach to model difference representation. Within this approach, the detected model differences are represented as a difference model, which conforms to a metamodel that is automatically derived from the metamodel of the to-be-compared base models. Such a derivation process itself is a model transformation. Also, because the base models and the difference models are all model artifacts, other model-to-model transformations are induced to compose models (e.g., apply a difference model to a base model to produce the other base model). Thus, such an approach can be supported in a modeling platform and does not require other ad-hoc tool support. A possible future improvement to DSMDiff would be to integrate this approach to assist in representation of model differences.

CONCLUSION

In this paper, we have defined the model differentiation problem in the context of Domain-Specific Modeling. The main points include: 1) domain-specific modeling is distinguished from traditional UML modeling because it is a variable-metamodel approach whereas UML is a fixed-metamodel approach; 2) the underlying metamodeling mechanism used to define a DSML determines the properties and structures of domain-specific models; 3) domain-specific models may be formalized as hierarchical graphs annotated with a set of syntactical information. Based on these characteristics, model differentiation algorithms and an associated tool called DSMDiff were developed to discover the mappings and differences between any two domain-specific models. The paper also describes our investigation into a visualization technique to display model differences structurally and highlight them using color and icons. The applicability of DSMDiff has been discussed within the context of model transformation testing. To conclude, the major contribution of this work is to provide efficient algorithms and a practical tool that identifies differences between domain-specific models, which is critical to many model development and management activities.

The project website for DSMDiff, which contains video demonstrations and the GME plug-in, can be found at *http://www.cis.uab.edu/gray/Research/DSMDiff*.

ACKNOWLEDGEMENT

This work was supported in part by an NSF CAREER award (CCF-0643725).

REFERENCES

ALANEN M and PORRES I (2003) Difference and union of models. In *Proceedings of the 6th International Conference on Unified Modeling Language (UML)*, pp 2-17, Springer-Verlag, LNCS 2863, San Francisco, California.

BUDINSKY F, STEINBERG D, MERKS E, ELLERSICK R, and GROSE T (2004) *Eclipse Modeling Framework*, Addison-Wesley, Redwood City, California..

CHAWATHE SS, RAJARAMAN A, GARCIA-MOLLINA H, and WIDOM J (1996) Change detection in hierarchically structured information. In *Proceedings of SIGMOD International Conference on Management of Data*, pp 493-504, ACM Press, Montreal, Canada.

CICCHETTI A, DI RUSCIO D, and PIERANTONIO A (2007) A Metamodel-independent approach to difference representation. *Journal of Object Technology* (Special Issue from *TOOLS Europe 2007*), June 2007, 20 pages.

EICK SG, STEFFEN JL, and SUMMER EE (2001) SeeSoft--A Tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering* 18(11), 957-968.

GRAY J, TOLVANEN JP, KELLY S, GOKHALE A, NEEMA S, SPRINKLE J (2007) Domain-specific modeling. In *Handbook of Dynamic System Modeling*, (Paul Fishwick, ed.), Chapter 7, CRC Press, Boca Raton, Florida.

HUNT JW and MCILROY MD (1975) An Algorithm for Differential File Comparison. Computing Science Technical Report, No. 41, Bell Laboratories.

KHULLER S and RAGHAVACHARI B (1996) Graph and network algorithms. *ACM Computing Surveys* 28(1), 43-45.

KURTEV I, BEZIVIN J, JOUAULT F and VALDURIEZ P (2006) Model-based DSL frameworks. In *Companion of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp 602-616, Portland, Oregon.

LEDECZI Á, BAKAY A, MAROTI M, VOLGYESI P, NORDSTROM G, SPRINKLE J, and KAISAI G (2001) Composing domain-specific design environments. *IEEE Computer* 34(11), 44-51.

LIN Y, ZHANG J and GRAY J (2005) A framework for testing model transformations. In *Model-Driven Software Development*, (Beydeda, S., Book, M. and Gruhn, V., eds.), pp 219-236, Springer, Berlin, Germany.

MAGICDRAW (2007) *Magic Draw UML*. http://www.magicdraw.com/

MANDELIN D, KIMELMAN D and YELLIN D (2006) A Bayesian approach to diagram matching with application to architectural models. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pp 222-231, Shanghai, China.

MEHRA A, GRUNDY J and HOSKING J (2005) A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE)*, pp 204-213, Long Beach, California.

METAEDIT+ (2007) *MetaEdit+ 4.5 User's Guide*. http://www.metacase.com

MICROSOFT (2005) *Visual Studio Launch: Domain-Specific Language (DSL) Tools: Visual Studio 2005 Team System*. http://msdn.microsoft.com/vstudio/teamsystem/workshop/DSLTools

OHST D, WELLE M and KELTER U (2003) Differences between versions of UML diagrams. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering*, pp 227-236, Helsinki, Finland.

ROSE (2007) *IBM Rational Rose*. http://www-306.ibm.com/software/awdtools/developer/rose/

SENDALL S and KOZACZYNSKI W (2003) Model transformation – The heart and soul of model-driven software development. *IEEE Software* 20(5), pp. 42-45.

WANG Y, DEWITT DJ and CAI J (2003) X -Diff: An effective change detection algorithm for XML documents. In *Proceedings of the 19th International Conference on Data Engineering*, pp 519-530, Bangalore, India.

XING Z and STROULIA E (2005) UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE)*, pp 54-65, Long Beach, California.

BIOGRAPHIES

Yuehua ("Jane") Lin is a senior software engineer at Honda Manufacturing of Alabama, LLC. Her research interests include model transformation and supporting tools. Jane received a Ph.D. in Computer Science from the University of Alabama at Birmingham (UAB).

Jeff Gray is an Associate Professor of Computer and Information Sciences at UAB where he co-directs research in the Software Composition and Modeling (SoftCom) laboratory. Jeff received a Ph.D. in Computer Science from Vanderbilt University. His research interests include model-driven engineering, model transformation, aspect-oriented development, and generative programming.

Frédéric Jouault is a postdoctoral researcher at UAB. He received his Ph.D. in Computer Science from the University of Nantes. His research interests include model engineering and model transformation with application to Domain-Specific Languages and model-based legacy reverse engineering. He is leading the development of the ATL language and toolkit.