Model Comparison with GenericDiff

Zhenchang Xing
School of Computing
National University of Singapore
xingzc@comp.nus.edu.sg

ABSTRACT

This paper proposes *GenericDiff*, a general framework for model comparison. The main idea is to separate the specification of domain-specific model properties and syntax from the general graph matching process and to use composite numeric vectors and pairup graph to encode the domain-specific properties and syntax so that they can be uniformly exploited in the general matching process. Our initial evaluation demonstrates that it is easy to deploy *GenericDiff* in a new application domain and *GenericDiff* is able to produce an accurate comparison reports for diverse types of models.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – Representation; E.1 [**Data Structures**]: Graphs and networks; F2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – Pattern Matching

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Model differencing, Graph matching, Metamodel

1. INTRODUCTION

Comparing artifacts and detecting their differences is an important operation in many application domains, such as software reuse and evolution, program debugging and fault localization, malware detection, and service composition. As a software system is often abstracted in models, a large number of domain-specific model comparison algorithms [5][9][14] have been proposed, tailored for a specific matching problem and model representation in different application domains. Such algorithms can be fairly accurate and efficient for a given application domain. However, due to the diversity of matching problems and model properties/syntax, the heuristics developed for one application domain cannot be reused in another.

To avoid building differencing techniques for new application domains from scratch, many exact and approximate graph matching algorithms have been proposed for the problem of graph isomorphism and its variants [2]. These algorithms can be applied to a wide class of models that can be represented as graphs. However, they are less efficient, since the general problem of graph isomorphism is NP-complete. Furthermore, due to the lack of the integration of domain knowledge in the matching process, they often produce a matching report that does not correspond well to the domain intuition.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'10, September 20–24, 2010, Antwerp, Belgium.
Copyright 2010 ACM 978-1-4503-0116-9/10/09...$10.00.

In this paper, we present our *GenericDiff* framework for model comparison. *GenericDiff* strikes a balance between being domain independent yet aware of domain-specific model properties and syntax. In the next section, we discuss the key ideas of *GenericDiff* framework. In Section 3, we demonstrate how we adapt *GenericDiff* to compare product feature models and discuss the limitations of the current implementation of *GenericDiff* in comparison with UMLDiff [14], a domain-specific algorithm for comparing UML class models.

2. GENERICDIFF FRAMEWORK

Figure 1 presents the architecture of *GenericDiff*. *GenericDiff* takes as input two models to be compared and the specifications of model properties and syntax in terms of domain-specific properties, pairup feasibility predicates, and random walk tendency functions. The generic matching process of *GenericDiff* casts the problem of comparing two models as the problem of recognizing the *Maximum Common Subgraph* (*MCS*) [1] of two Typed Attributed Graphs (TAGs). It uses two data structures, i.e., composite numeric vector and pairup graph, which allows domain-specific model properties and syntax to be uniformly exploited in the matching process. *GenericDiff* outputs a symmetric difference between two compared models, which can serve as input to different application domains.

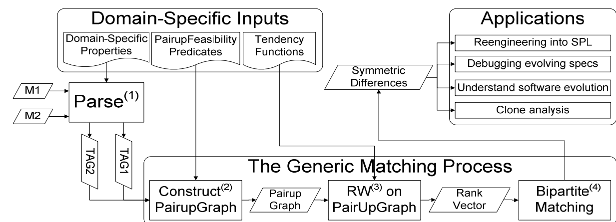

Figure 1 The architecture of GenericDiff

Given two models to be compared, M_1 and M_2, *GenericDiff* parses[(1)] the input models into two *Typed Attributed Graphs* (*TAGs*), according to the metamodel of M_1 and M_2. The types of graph nodes and edges correspond to the metaclasses and metaassociations defined in the metamodel. Given a meta-model, one needs to select a set of metaproperties for each type of model element and relation that characterize its instances. During the parsing process, *GenericDiff* collects data from the selected properties of each model element and relation and represents them in a composite numeric vector attribute, associated with the corresponding graph node and edge. *GenericDiff* defines a set of atomic numeric vectors, together with standard distance calculators, for commonly used data types. The atomic vectors can then be composed into composite vectors. This composite vector attribute is a compact

representation of the properties of model elements and relations for efficient graph indexing and matching.

Given two typed attributed graphs, $TAG_1(V_1,E_1)$ and $TAG_2(V_2, E_2)$, of two compared models, *GenericDiff* constructs a *PairupGraph* [1] $PUG(V_{pu},E_{pu})$, which is a product of two typed attributed graphs, i.e., $V_{pu} \subseteq V_1 \times V_2$ and $E_{pu} \subseteq E_1 \times E_2$. A node (edge) of *PUG* represents a pair of nodes (edges) of two *TAGs*. PairupGraph captures the graph structure of two compared models. The construction of PairupGraph is guided by a set of user-defined pairup feasibility predicates. These predicates help to prune search space according to domain-specific knowledge. There are two kinds of feasibility predicates, respectively regarding the domain-specific properties and syntax of the compared models. These feasibility predicates define the type compatibility, minimum attribute similarities and the topological constraints (e.g., containment) that a pair of graph nodes (edges) must satisfy so that they can be paired-up. The current implementation of *GenericDiff* supports only the property-based feasibility predicates.

The initial distance value of a PairupGraph node (edge) is calculated as the Euclidean length of the normalized distance vector between the two paired-up graph nodes (edges). *GenericDiff* performs a random walk(3) [10] on the PairupGraph, which is an iterative process that propagates the distance values from node pair to node pair based on graph structure. A random walk on a graph *G* can be described by a probabilistic model that allows us to compute the probability $r_n(t)$ of being located in each node *n* at time step *t*. The probability distribution on all the nodes is represented by a vector $r(t) = [r_1(t), \ldots, r_N(t)]$, *N* being the number of nodes in the graph. The probabilities $r_n(t)$ are updated at each time step as follows:

$$r_n(t+1) = \sum_{s \in G} jump(s,n) \times jump(s) \times r_s(t) + \sum_{s \in src(n)} follow(e^{s \to n}) \times follow(s) \times r_s(t)$$

where $jump(s,n)$ and $follow(e^{s \to n})$ are the probabilities of moving from node *s* to node *n* by jumping or by following an edge, respectively, and $jump(s)$ and $follow(s)$ represent the bias between these two possible actions. These parameters describe the behavior of random walk. By default, *GenericDiff* assumes that $jump(s)$ and $follow(s)$ are independent of the PairupGraph node *s* for a random walk on PairupGraph. Thus, *GenericDiff* considers a parameter $df \in (0,1)$ such that $follow(s) = df$ and $jump(s) = 1 - df$. Furthermore, *GenericDiff* provides four sets of default random walk tendency functions for defining $jump(s,n)$ and $follow(e^{s \to n})$ as constant functions, functions of the distance values of the PairupGraph nodes and edges, and functions of $r_n(t)$.

The random walk on the PairupGraph outputs a rank vector of graph node pairs, each of which is assigned a numerical correspondence measure, i.e., the measure of the quality of the match it represents. *GenericDiff* constructs a bipartite graph from this rank vector of node pairs and selects an optimal matching(4) using stable-marriage algorithm (Gale-Shapley algorithm) [3]. A bipartite matching is stable if there are no two entities that would both prefer each other over the entities that they are currently matched with. Finally, given a pair of matched graph nodes, *GenericDiff* builds a bipartite graph of their edges and uses the stable-marriage algorithm again to map their edges.

3. EVALUATION

We have applied *GenericDiff* in several applications for the purpose of detecting feature variants in a software product family, debugging evolving behaviors of formal specifications, recognizing evolutionary changes to a program, and identifying differences between clone instances. In these applications, *GenericDiff* has been applied to compare product feature models, labeled transition systems, UML class models, and program dependence graphs, respectively. In this section, we illustrate how we adapt *GenericDiff* to compare product feature models. We also comparatively evaluate the limitations of the current implementation of *GenericDiff* against UMLDiff [14], a domain-specific differencing algorithm for UML class models

3.1 Extractive reengineering into SPL

Product variants are often created by copy-paste-modify various parts of the existing products that have been successfully developed for some customers. Such product variants are often a starting point for building Software Product Line (SPL), i.e., extractive reengineering into SPL. The first step in this extractive approach is therefore to understand common and variant features in existing product variants. This problem is non-trivial for a big software product family with many features that has been evolved for long time. During evolution, feature names and descriptions as well as the dependencies and relationships between features might have been changed. New features might have been added and existing features might have been deleted. The features might have also been split or merged.

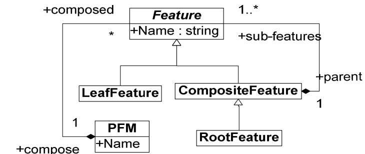

Figure 2. The meta-model of PFM

We have applied *GenericDiff* to assist analysts in detecting changes to product features during the evolution of product variants. We first entail that features and their inter-dependencies for each product variant are documented as Product Feature Model (PFM). Figure 2 presents the meta-model of product feature model. A PFM forms a hierarchy of product features. Each *Feature* in a PFM must be uniquely identifiable by its *name* property, which can be any free-form text that describes the feature. A PFM must have a *RootFeature*, which is a special *CompositeFeature* that represents the corresponding product variant. A *Feature* can be decomposed into sub-features. A feature that has no sub-features is a *LeafFeature*; otherwise it is a *CompositeFeature*. The root feature has no *parent* feature, while a non-root feature must have a *parent* feature, i.e. belongs to a composite feature.

Given the PFMs $\{PFM_1..PFM_N\}$ of *N* product variants, we apply *GenericDiff* to compute pair-wisely the differences between the product feature models PFM_i and PFM_j $(i \neq j; 1 \leq i,j \leq N)$ of two product variants. The TAG of a PFM forms a *containment tree*, which consists of three types of graph nodes, corresponding to the metaclass *RootFeature*, *LeafFeature* and *CompositeFeature* respectively. Graph edges represent the instance of *parent-subfeature* metaassociation between features.

A feature in PFM has three properties, i.e., *name*, a *parent* feature, and a (possibly empty) set of *subfeatures*. We define,

for a feature node *f*, a composite vector of two atomic vectors. One atomic vector represents the set of words in the *name* property of *f*. We choose the Jaccard coefficient to measure the similarity between two sets of words S_1 and S_2, i.e., $S_1 \cap S_2 / S_1 \cup S_2$. The other atomic vector is a numeric vector that stores the number of the *subfeatures* of *f*. Given two such numeric vectors, $[v]$ and $[v']$, we choose Manhattan (Taxicab) distance, i.e., $|v - v'|$, to measure their similarity. The edges of the containment tree of a PFM have no characteristic vectors, since they simply represent the *parent-subfeature* relations between features.

For the comparison of PFMs, we define one property-based feasibility predicate, i.e., the type compatibility of graph nodes [*Feature*, *Feature*]. Note that we define the type compatibility in terms of the supertype *Feature*. That is, the mappings between graph nodes of different subtypes, such as *LeafFeature* and *CompositeFeature*, are allowed in this empirical study. For the comparison of PFMs, we set *df* at 0.85. We use the default random walk tendency function that define $jump(s, n)$ and $follow(e^{s \to n})$ as linear functions of the distance values of the relevant PairupGraph nodes and edges.

GenericDiff reports a symmetric difference between two compared PFMs, PFM_1 and PFM_2, i.e., a set *M* of corresponding (i.e., matched) features that exist in both PFMs, a set UM_1 of features that are unique in PFM_1, and a set UM_2 of features that are unique in PFM_2. Based on the differencing report by *GenericDiff*, we developed a tool for automatically inferring six types of feature changes that can evolve PFM_1 into PFM_2 based on the effects of feature changes on the PFMs. These feature changes include: renaming feature, add leaf feature and add feature subtree, remove leaf feature and remove feature subtree, move feature, split feature and merge feature.

We evaluated the accuracy and scalability of *GenericDiff* using a real-world product family of financial systems (WingSoft Financial Management Systems) as well as a large volume of systematically synthesized data. Overall, *GenericDiff* yields accurate change reports between the PFMs of product variants, even the PFMs have undergone intensive evolution. We manually inspected our empirical data and identified two main causes of false positive (i.e., erroneously reported) changes and false negative (i.e., missed) changes using *GenericDiff*.

First, it is difficult for *GenericDiff* to determine the mappings between features with little or very similar structural context (i.e., dependencies and relationships with other features). For example, the leaf features become an issue, since they have no structural information other than their name property. Second, if product features suffer various types of changes at the same time, for example, a feature *f* is moved to another composite feature, and then split into several features, finally it is renamed. *GenericDiff* may not recognize the correspondences between the origin feature *f* and the resulting feature, since their name properties and structural context changed dramatically.

3.2 Comparison with UMLDiff

To comparatively evaluate *GenericDiff* with a domain-specific differencing algorithm, we applied *GenericDiff* to the empirical data used to evaluate UMLDiff [14], a differencing algorithm for comparing UML class models. The data set consists of the reversed-engineered class models of 11 releases of HtmlUnit, a unit testing framework for web applications, and of 31 releases of JFreeChart, a Java library for drawing charts. The GenericDiff's precision and recall in recognizing renamed and moved program entities is 69% and 84% for JFreeChart and 84% and 42% for HtmlUnit. The precision and recall of UMLDiff is 91% and 93% for JFreeChart and 92% and 95% for HtmlUnit [14] respectively. We manually inspected the comparison report of *GenericDiff* against that of UMLDiff, which revealed that the main causes for the poorer quality of the GenericDiff's comparison report are due to three limitations of current implementation of *GenericDiff*. Since the current implementation of *GenericDiff* suffers the similar symptoms in two subject systems, we explain these causes using mainly JFreeChart data in this section. It is important to note that these limitations are not fundamental flaws of the proposed *GenericDiff* approach. We are currently investigating several enhancements to the implementation of *GenericDiff*. As discussed below, these enhancements can be easily integrated in the architecture of *GenericDiff* (see Figure 1) and they can significantly improve the quality of *GenericDiff*.

3.2.1 Matching thresholds

UMLDiff uses the user-specified renaming and move thresholds: two entities are considered as renaming or move candidates only if their similarity metric is above the thresholds. As reported in [14], the quality of UMLDiff's change report relies on the specification of appropriate thresholds, which is a highly non-trivial task. The higher the threshold is, the stricter the similarity requirement is. The smaller the threshold is, the riskier the renaming and moves are. For example, the precision of UMLDiff degrades to 73% at the renaming and move threshold 0.01 in the JFreeChart case study, which is similar to the precision (69%) of *GenericDiff* for JFreeChart.

In contrast, *GenericDiff* does not rely on such arbitrary thresholds. We are now investigating the use of statistical criterion, such as quartile or Chauvenet [12], to remove the matching outliers in the comparison report of *GenericDiff*. For example, 35% of the erroneous matches in JFreeChart case can be removed with the lower quartile 25%. In the case that the user is able to specify appropriate matching thresholds, *GenericDiff* can take as input such domain-specific heuristics in terms of pairup feasibility predicates, such as minimum attribute similarities. These feasibility predicates can help to filter out the pair-ups of irrelevant nodes (edges).

3.2.2 Topological restrictions

The two subject systems have been under active development and they have suffered a substantial amount of changes between releases [14]. UMLDiff exploits the containment-spanning tree of UML class models to prune the set of matching candidates. For example, it only considers potential renaming candidates within the context of two already mapped parent entities. Furthermore, the reverse-engineered class models consist of a large volume of entities with little structural information. For example, about 1/4 of all the methods have no incoming usage dependencies in the JFreeChart data; about 1/3 of the methods have no outgoing usage dependencies. UMLDiff exploits the domain-specific syntax, such as inheritance hierarchy and the similarity of containing entities to determine the correspondences between entities with little structural context.

In contrast, the currently implementation of *GenericDiff* is not aware of the presence of the containment-spanning tree in the graph of UML class model. Thus, it tends to overly pair-up matching candidates. Furthermore, the entities with little

structural context also pose a threat to the accuracy of *GenericDiff*. For example, those methods without incoming and/or outgoing usage dependencies account for 56% of all the erroneous matches and 37% of all the missed matches in the JFreeChart study. We are extending the implementation of *GenericDiff* to support the syntactic pairup feasibility predicates, which can be utilized to specify domain-specific topological restrictions, such as the containment and inheritance relations in UML class models.

3.2.3 Matching cardinality

UMLDiff reports many-to-one matching between moved entities, while the currently implementation of *GenericDiff* report only one-to-one matching, due to the limitation of Gale-Shapley algorithm [3]. Thus, *GenericDiff* currently misses many-to-one moves. For example, two new superclasses ClickableElement and StyledElement were introduced into HtmlUnit in the version 1.3. All the 918 getAttributes() methods that used to scatter in 65 existing subclasses were pulled-up into the new superclasses. *GenericDiff* missed 860 of these method moves, which accounts for 81% of all the missed matches in the HtmlUnit study. We are now extending the *GenericDiff* implementation to integrate new bipartite matching algorithms, such as [6] that are able to find many-to-one matching.

3.2.4 Other domain-specific assumptions

UMLDiff makes some other assumptions about the potential correspondences between entities. For example, it only reports the moves of same-name entities; it assumes that the same-name entities contained in a pair of mapped entities are matched. In contrast, *GenericDiff* imposes no constraints on what entities may be considered as matched, renamed and/or moved. This tends to results in an issue of over-pairup, especially when it compares the releases in which a large amount of entities has been changed, added, and removed during evolution. To improve the accuracy of *GenericDiff*, we are now intestigating the use of description logic to specify these heuristics or assumptions as domain-specific inputs to *GenericDiff*.

4. RELATED WORK

Only a few generic model comparison algorithms and tools have been proposed so far. One such algorithm is SiDiff [13]. Similar to *GenericDiff*, the data model of SiDiff is also typed attributed graph. However, SiDiff still requires the presence of a primary structure (e.g., a containment tree) in the compared models; and it still relies on the ad hoc matching weights assigned to node attributes to determine the similarity. The EMF Compare [16] is another generic matching engine. It compares Ecore models. EMF Compare is metamodel agnostic and its matching strategy is very close to the domain-specific algorithm UMLDiff [14].

In contrast, the matching process of *GenericDiff* is general. *GenericDiff* exploits many concepts and techniques developed in general graph matching, including the modular product of two graphs for solving maximum common subgraph problem [1], the reformulation of graph matching into a considerably simpler bipartite matching problem [11], and the application of spectral analysis to encode important structural properties of the graph [4]. Furthermore, inspired by Google's PageRank [10], *GenericDiff* employs an iterative distance propagation process to spread the distance value in the PairupGraph. Several other algorithms [7][8] have also been proposed for measuring the similarity between elements based on random walk. Finally, similar to other graph-based techniques [13][15], *GenericDiff* also seeks a reduced representation for efficient graph indexing and matching; it encodes the domain-specific properties of model elements and relations in composite numeric vectors.

5. CONCLUSION AND FURTURE WORK

This paper proposes *GenericDiff*, a general framework for model comparison. Out initial evaluation demonstrates that: because it separates the specification of domain-specific information from the general matching process, it is easy to deploy *GenericDiff* in a new application domain; because it leverages the concepts and techniques developed in general graph matching, it can produce an accurate comparison report for diverse types of models. Future work on *GenericDiff* includes addressing the limitations of current *GenericDiff* implementation revealed in our case studies. We also plan to apply *GenericDiff* to compare heterogeneous models, i.e., a mixture of different types of models.

ACKNOWLEDGEMENT

This work has been supported by the Ministry of Education of Singapore (AcRF Tier 1 R-252-000-399-112) and by the National Sciences and Engineering Research Council of Canada.

REFERENCES

[1] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM* 16:575-577.

[2] D. Conte et al. Thirty years of graph matching in pattern recognition. *IJPRAI*, pp. 265-298, 2004.

[3] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Math.*, 69:9-14, 1962.

[4] M. Gori and M. Maggini. Exact and appropriate graph matching using random walk. *TPAMI'05* 27(7):1100-1111.

[5] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *PLDI'90*, pp. 234-246.

[6] R.W. Irving et al. The hospitals/residents problem with ties. *LNCS* 1851, pp. 259-271, 2000.

[7] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. *KDD'02*, pp. 538-543.

[8] S. Melnik, H.C. Molina, E. Rahm, Similarity flooding: A versatile graph matching algorithm and its application to schema matching. *ICDE'02*, pp. 177-186.

[9] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook and P. Zave. Matching and merging startcharts specifications. *ICSE'07*, pp. 54-64.

[10] L. Page et al. The PageRank citation ranking: Brining order to the web. Technical Report, Stanford Info Lab, 1999.

[11] K. Riesen et al.. Efficient suboptimal graph isomorphism. *GbRPR'09*, pp. 124-133.

[12] J. Taylor. *An Introduction to Error Analysis*. 2nd edition. University Science Books, pp 166-168, 1997.

[13] C. Treude, S. Berlik, S. Wenzel and U. Kelter. Difference computation of large models. *ESEC/FSE'07*, pp. 295-304.

[14] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. *ASE'05*, PP. 54-65.

[15] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. *Int. Conf. on Data Mining*, pp. 721, 2002.

[16] EMF: http://www.eclipse.org/emft, 2010.