# GenericDiff: A General Framework for Model Comparison

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

## Abstract

This paper presents *GenericDiff*, a general framework for model comparison. The main design challenge of *GenericDiff* lies in how to strike a balance between being domain independent yet aware of domain-specific model properties and syntax. *GenericDiff* tackles this challenge by separating the specification of domain-specific inputs from the generic graph matching process and by making use of two data structures, i.e., typed attributed graph and pairup graph, to encode the domain-specific properties and syntax so that they can be uniformly exploited in the generic matching process. Comparing large models efficiently is another challenge. *GenericDiff* leverages two techniques, i.e., random walk on graph and bipartite graph matching, to efficiently compute a difference between models. To date, *GenericDiff* has been deployed in three applications to compare UML class models, Labeled Transition Systems, Program Dependence Graphs and Feature Models. These applications demonstrate that it is easy to adapt *GenericDiff* in a new application domain and *GenericDiff* is able to produce an accurate comparison report for diverse types of models.

## Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks; F2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithm and Problems – Pattern Matching

## General Terms

Algorithms, Design, Experimentation

## Keywords

Model differencing, Graph matching, Metamodel

## 1. Introduction

Comparing artifacts and detecting their differences is an ubiquitous operation, relevant in many application domains, such as software reuse and evolution [18,20,24,30,47], debugging and fault localization [33,35,39], malware detection [6], and service integration [5,52]. It can be present in diverse forms, such as detecting variants in a software product family, recognizing changes to a program, debugging evolving behaviors of formal specifications, identifying deviations of API usage in applications, and detecting inconsistencies between interacting services. As a software system is often abstracted in models, a large number of model comparison algorithms [18,20,24,30,31,33,41,47] have been proposed, tailored for a specific matching problem and model representation in different application domains. These algorithms are highly tuned with domain-specific heuristics, such as topological restrictions and domain-specific properties.

For example, UMLDiff [47], designed for comparing UML class models, traverses the containment-spanning tree of two class models and identifies corresponding entities based on their name and structure (e.g. inheritance and usage dependency) similarity. As another example, Nejati et al. [30] propose a matching algorithm for statechart specifications that determines how close the behaviors of one state are to those of another based on the state and transition labels and the approximate bisimilarity of states. 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. Differencing techniques for new application domains must usually be built from scratch, which requires significant amount of thought and effort.

To avoid investing such effort for each new domain where a comparison algorithm is required, many exact and approximate graph matching algorithms have been proposed for the general problem of graph isomorphism and its variants [8]. For example, Bron and Kerbosh [4] use tree search approach to find the *Maximum Common Subgraph* (MCS) of two graphs in a suitably defined association graph. Shokoufandeh and Dickinson [40] utilize random walk on graph to obtain a discriminating index of graph structure. Riesen et al. [37] present an efficient suboptimal graph isomorphism algorithm based on bipartite graph matching. These algorithms can be applied to a wide class of models that can be represented as graphs. But they are usually less efficient, since the general problem of graph isomorphism is NP-complete [8]. Furthermore, due to the lack of the integration of domain-specific knowledge in the matching process, they often produce a matching report that does not correspond well to the domain intuition.

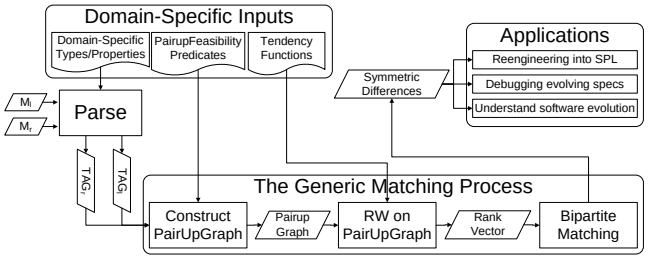


**Figure 1 The architecture of *GenericDiff***

In this paper, we present *GenericDiff*, a general framework for software model comparison. As shown in Figure 1, *GenericDiff* takes as input two models to be compared and the specifications of model properties and syntax in terms of domain-specific types/properties, pairup feasibility predicates and random walk tendency functions. It casts the problem of comparing two models as the problem of recognizing the MCS of two Typed Attributed Graphs (TAGs).

Given two models to be compared, $M_l$ and $M_r$, *GenericDiff* parses the input models into TAGs. It encodes the domain-specific properties in composite vector attributes to quantify the corresponding graph nodes and edges. *GenericDiff* then constructs a PairUpGraph [4], i.e., a product of two TAGs, to capture the graph structure of two input models. After that, it performs a random walk on PairUpGraph, which is a probabilistic iterative process that propagates the correspondence value from node pair to node pair based on graph structure. The random walk on PairUpGraph outputs a rank vector of graph node pairs, each of

which is assigned a quantitative correspondence value, i.e., a measurement of the quality of the match it represents. *GenericDiff* builds a bipartite graph from this rank vector of node pairs and selects an optimal matching using bipartite matching algorithms [14,19]. *GenericDiff* outputs a symmetric difference between two input models, which serve as input for domain-specific analysis.

We have implemented *GenericDiff* and deployed it in three applications to compare product feature models [23], program dependence graphs (PDGs) [11], labeled transition systems (LTSs) [7] and UML class models [54]. These models have distinct model properties and syntax. They are widely adopted for describing the requirements, structure and behavior of software systems. In this paper, we illustrate how *GenericDiff* has been adapted to: 1) compare UML class models for understanding the evolution of software design; 2) compare LTSs for debugging changing behaviors of a CSP# [42] specification; and 3) compare PDGs for characterizing the differences of software clones. These applications demonstrate that it is easy to deploy *GenericDiff* in a new application domain – it took only a few trials for a domain expert (who is not necessarily familiar with model and graph matching techniques) to develop the necessary domain-specific inputs, which is the only step required to apply *GenericDiff* to diverse types of models. Furthermore, *GenericDiff* effectively exploits these domain-specific inputs in its generic matching process and produces a comparably accurate comparison report to those of domain-specific differencing algorithms.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents our *GenericDiff* framework. Section 4 demonstrates the adaptation of *GenericDiff* for the comparison of UML class models and LTSs. Finally, we conclude and sketch possible future research directions.

## 2. Related work

The domain-specific model differencing techniques exploit the domain-specific topological restrictions and properties to traverse the mode graph and determine the corresponding entities. Researchers have presented techniques to compare class model [31,47], state machine [30,41], program dependency graph [18,20]. Only a few generic model comparison algorithms and tools have been proposed. One such algorithm is SiDiff [44]. Similar to *GenericDiff*, the data model of SiDiff is also typed attributed graph. However, SiDiff requires the presence of a primary structure (e.g., a containment tree) in the models to be compared; and it relies on the ad hoc matching weights assigned to node attributes to determine the similarity. The EMF Compare [53] is another generic matching engine. It compares Ecore models. EMF Compare is metamodel agnostic and its matching strategy is very close to UMLDiff [47], an algorithm for comparing UML class model.

Our position paper [48] proposed *GenericDiff*. In this paper, we provide a full description of the *GenericDiff* framework. The generic matching process of *GenericDiff* does not assume any domain-specific properties and graph structure. Given a domain-specific metamodel, *GenericDiff* can be easily adapted by developing the necessary specifications of model properties and syntax. This is an easier task than determining ad hoc edit costs or matching weights, since the specification of these domain-specific inputs corresponds straightforwardly to the underlying metamodel. Furthermore, instead of selecting mappings based on absolute similarity metrics and an arbitrary cutoff threshold, *GenericDiff* uses bipartite matching algorithms [14,19] that requires only the relative ranking of candidate pairs to select an optimal subset of matched pairs from a list of ranked candidates.

Our recent works [50] presented how we applied *GenericDiff* to compare product feature models [23] for analyzing feature variants in a family of similar software products. In this paper, we present three new applications of *GenericDiff* to the comparison of three types of more complicated models, i.e., UML class models, labeled transition systems and program dependence graphs.

Exact and approximate graph matching has been studied for decades [8]. Recently, graph-based techniques have been receiving a growing attention from the scientific community, due to the fact that the computational cost of the graph-based algorithms, although still high, is now becoming compatible with the computational power of new computer generations [8]. *GenericDiff* exploits many concepts and techniques developed in general graph matching, such as the modular product of two graphs for solving maximum common subgraph problem [4], the first-order feasibility rules to capture the matching constraints [9], the reformulation of graph matching into a considerably simpler bipartite matching problem [37], the application of random walk to encode graph topology [15,40], and the propagation of local constraints to neighboring nodes by an iterative process [46].

Most model comparison algorithms, including *GenericDiff*, examine not only the local properties of two entities but also their structural context. The underlying intuition is that, similar entities are related to other similar entities. Unlike previous approaches [44,47,53] that examine only immediate common neighbors, *GenericDiff*, inspired by PageRank [32], employs a random walk on PairUpGraph to spread the correspondence value in the PairUpGraph. PageRank is an iterative link propagation and analysis algorithm based on random walk on graph [27]. Inspired by PageRank, several algorithms [21,28] in data mining domain have also been proposed for measuring the similarity between elements based on random walk on graph. However, these algorithms examine only graph structure. They do not provide systematic ways to encode domain-specific properties.

Graph-based techniques often seek a reduced representation for efficient indexing and matching. For example, in [15,40], the eigenvectors of a graph's adjacency matrix have been used to encode important structural properties of the graph. The clone miner ModelCD [34] uses a numeric vector representation to approximate graph paths. gSpan, a subgraph pattern miner [51] encodes graph structures in depth-first search subscripts. SiDiff [44] represents an element in a collection of numeric metrics. *GenericDiff* encodes the domain-specific properties of model elements and relations in composite vector attributes associated with the corresponding nodes and edges of model graphs.

## 3. GenericDiff Framework

In this section, we first justify the rationale behind the design of *GenericDiff* from the perspective of metamodeling (Section 3.2). We then present two data structures, i.e., Typed Attributed Graph (Section 3.3) and PairUpGraph (Section 3.4), that allows *GenericDiff* to effectively encode domain-specific model properties and syntax in a systematic, domain-independent way. In Section 3.5, we discuss the random walk on PairUpGraph that propagates the correspondence value from node pairs to neighboring node pairs by an iterative process. In Section 3.6, we discuss how *GenericDiff* reformulate the maximum common subgraph problem to a considerably simpler bipartite matching problem for which polynomial algorithms exist. We also discuss how *GenericDiff* reduces the loss of graph-structure information in

this problem reformulation. Finally, we discuss the output of *GenericDiff* and its time complexity.

## 3.1 A running example

We demonstrate the key components of *GenericDiff* with the running example shown in Figure 2. The Auctioneer and Bidder are two interacting protocols that are supposed to coordinate to complete the bidding process. Clearly, the two protocols are incompatible. In addition to incompatible messages, there exist two more complicated behavior inconsistencies. The auctioneer starts a bid by sending out *newItem* message and then waits for new bids or requests for update. The bidder, after receiving *auctionBegin* message, initiates a request for permission to join the bid. But the Auctioneer does not response to this request. Furthermore, the auctioneer sends out bid update on demand, but the bidder assumes that the auctioneer sends out update without explicit request. To detect these inconsistencies, we apply *GenericDiff* to compare the state models of the incompatible Auctioneer and Bidder protocols.

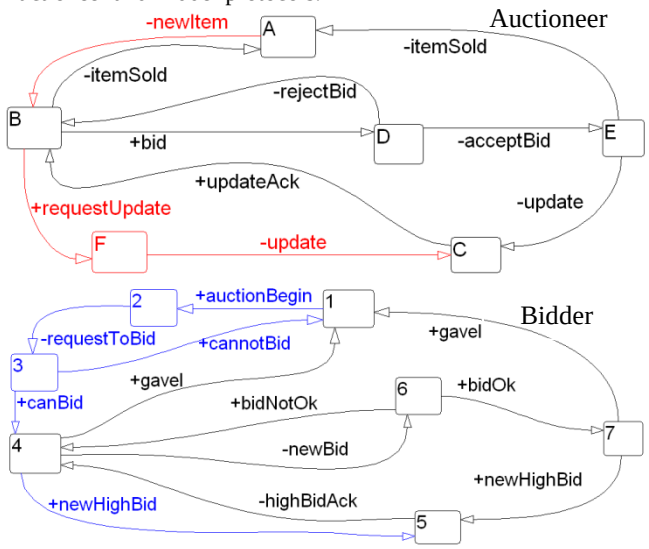


**Figure 2 The state models of Auctioneer and Bidder**

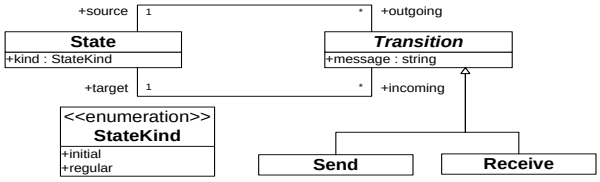


**Figure 3 The metamodel for interacting state models**

## 3.2 The metamodel

*GenericDiff* separates the specification of domain-specific model properties and syntax from the generic graph matching process. The rationale behind this design lies in the metamodeling capability of separating generic metadata from domain-specific metamodel. A domain-independent metamodel, such as the Meta Object Facility (MOF) 2 [54] or Eclipse Ecore [53], is used to model other metamodels and models. Four key modeling concepts are *Class*, *DataType*, *Association*, and *Property*. These modeling concepts provide a generic data management framework when they are used in two-level modeling, in which they define generic models. The generic models consist of fixed types of model elements and relations. For example, the state models of Auctioneer and Bidder, represented as generic models, consist of model elements and relations that are the instances of metaclass *Class* and *Association*.

Often, the specific types and properties of model elements and relations and the syntactic (i.e., structural) information of models are prescribed by a domain-specific metamodel in multi-level modeling. Figure 3 shows the metamodel for the state models in our running example. This metamodel defines that a state model consists of states and transitions, i.e., instances of metaclasses *State* and *Transition*. A state represents a situation during which the system waits for some external events to occur. A state owns a property of the enumeration type *StateKind*, which can be of literal values *initial* or *regular*. A state also has a collection of *incoming* transitions and a collection of *outgoing* transitions. A transition is a directed relation from a *source* state to a *target* state. There are two types of transitions, *Send* and *Receive*, which are marked with + and – signs in the Auctioneer and Bidder models, respectively. A transition has a *message* property that represents the message being sent or received. In the Auctioneer model, the initial state A (the state name/index is only for illustration purpose) transits to the regular state B by sending the message *newItem*. The transition *newItem* is an outgoing transition from the source state A and an incoming transition to the target state B.

## 3.3 Typed attributed graph

*GenericDiff* considers the input model as a generic model and thus internally represents it as a Typed Attributed Graph (TAG). The TAG is a generic representation of the input model, which makes the kernel of *GenericDiff* independent of specific types of models. A graph node (i.e., an instance of *Class*) of TAG represents a model element. A graph edge (i.e., an instance of *Association*) represents a relation between model elements. When parsing the input model into a TAG, *GenericDiff* encodes the specific type of model elements and relations (i.e., their metaclass defined in the domain-specific metamodel) in a type attribute of the corresponding graph nodes and edges. Furthermore, given a domain-specific metamodel, one needs to specify a set of metaproperties for each type of model elements and relations that characterize its instances. During the parsing process, *GenericDiff* collects the data from the selected properties of model elements and relations and represent them in a composite vector attribute of the corresponding graph nodes and edges.

*GenericDiff* defines a set of atomic vector representations, associated with standard distance calculators, for four basic data types, i.e., enumeration, numeric, string and collection (See Appendix 1). The atomic vectors can then be composed into composite vectors recursively, representing the domain-specific properties of structured datatypes. This composite vector attribute is a compact and uniform representation of the properties of model elements and relations for efficient graph indexing and matching.

Let $v_l$, $v_r$ be two atomic vectors, we denote their distance as $d(v_l,v_r)$, $d$ is the corresponding distance calculator defined in *GenericDiff*. Let $V_l$, $V_r$ be two composite vectors, we denote their distance as $d(V_l, V_r)=\sqrt{\sum_{i=1}^{n} d(V_l[i], V_r[i])^2}$, i.e., the Euclidean length of the distance vector $[d(V_l[i], V_r[i])]_{i=1..n}$, where $V_l[i]$ and $V_r[i]$ are the $i^{th}$ vector element of $V_l$ and $V_r$. $V_l$, or $V_r$ can be a null vector. When defining the vector attributes for selected metaproperties, one needs to specify the corresponding null atomic vectors, which can then be composed into composite null vectors. The null vectors represent the initial, undefined, or simplest state of a property. They can be (but not necessarily) an empty set, a zero-length sequence or a zero numeric vector.

In our running example, the TAG of the state model consists of graph nodes whose type attribute is *State* and graph edges whose type attribute is *Send* or *Receive*. We specify *kind*, *incoming* and *outgoing* as three characteristic properties of state. We define the composite vector representation for the three characteristic properties of state as [[kind], [|incoming|, |outgoing|]]. Thus, the composite vector attribute of a graph node consists of two atomic vectors. One is a literal index vector for the enumeration property *kind*. Since we are only interested in whether two literal values are different, hamming distance is selected to measure the similarity of two literal index vectors. Given two literal index vector $v_l$, and $v_r$, their hamming distance is $|\{i|v_l[i] \neq v_r[i]\}|$, i.e., the number of vector elements that are different. The other atomic vector is a numeric vector summarizing the size of the collection properties *incoming* and *outgoing*. We select Manhattan distance to measure the size differences between two collections. Given two numeric vector $v_l$, and $v_r$, their Manhattan distance is $\sum|v_l[i] - v_r[i]|$, i.e., the sum of the value differences of corresponding vector elements. We define the composite null vector of state as [[undefined],[0,0]].

According to the above specification of domain-specific properties, the vector attribute of state 2 is [[regular],[1,1]] since it is a regular state and has one incoming and one outgoing transition. The vector attribute of state B is [[regular],[3,3]]. The hamming distance of the first literal index vector of these two states is 0; the Manhattan distance of the second numeric vector is 4. The distance between the composite vector attribute of these two states is the Euclidean length of the distance vector [0,4], i.e. $\sqrt{0^2 + 4^2} = 4$. The distance of state 2 to null is the Euclidean length of the distance vector [1,2].

We specify *message* as the characteristic property of transition and select word set to encode the *message* property. The message string is split into a set of words based on case switching. For example, the message *newItem* is represented as [new, item]. We select Jaccard coefficient, a commonly used metric for measuring the similarity between sets of elements. Given two word sets $v_l$, and $v_r$, their Jaccard coefficient is $|v_l \cap v_r|/|v_l \cup v_r|$, i.e., the size of the intersection of two sets divided by the size of their union. The Jaccard coefficient between the messages *newItem* and *auctionBegin* is 0 since the two messages have no common words.

## 3.4 PairUpGraph

The composite vector attributes cannot always distinguish the instances of model elements. For example, one cannot tell whether state D of the Auctioneer model corresponds to state 3 or 6 of the Bidder model, since they all have the same composite vector attributes, [[regular],[1,2]]. The graph topology, i.e., structural context in which two elements appear can serve as another discriminating index that characterizes the model elements.

Given two TAGs $G_l(V_l,E_l)$ and $G_r(V_r,E_r)$, corresponding to two models to be compared, *GenericDiff* constructs a PairUpGraph $PUG(V_{pu},E_{pu})$ as follows:

$$([n_l, n_r], [n_l', n_r'] \in V_{pu}) \wedge ([e_l, e_r]^{[n_l,n_r]\to[n_l',n_r']} \in E_{pu})$$
$$\leftrightarrow (n_l, n_l' \in V_l) \wedge (e_l^{n_l \to n_l'} \in E_l) \wedge (n_r, n_r' \in V_r) \wedge (e_r^{n_r \to n_r'} \in E_r)$$
$$\wedge f(n_l, n_r) \wedge f(n_l', n_r') \wedge f\left(e_l^{n_l \to n_l'}, e_r^{n_l \to n_r'}\right)$$

A PairupGraph is a product of two input TAGs $G_l$ and $G_r$, i.e., $V_{pu} \subseteq V_l \times V_r$ and $E_{pu} \subseteq E_l \times E_r$ It captures the graph structure of two models. A node (edge) $p$ of PairUpGraph represents a pair of nodes (edges) ($p_l$, $p_r$) of two input TAGs. We define the initial distance value of $p$ as $d(p) = d(p_l, p_r) = d(V_l, V_r)$, where $V_l$, and $V_r$ are the composite vector attributes of $p_l$ and $p_r$ respectively.

As shown for the state, the selected characteristic properties can be of different data types and of different representations. Thus, the elements of the composite vectors may differ in their scales. They have to be normalized before the distance computation of two composite vectors. Given a set of distance vectors *DV* of a type of node (edge) pairs, let $dv_i$ be the $i^{th}$ dimension of a distance vector $dv \in DV$, the normalized value is given by subtracting the mean (mean-shifting) and then dividing the mean-shifted value by the standard deviation (auto-scaling):

$$dv_i = \frac{dv_i - mean(\{x_i | x \in DV\})}{stddev(\{x_i | x \in DV\})} \quad (1)$$

After mean-shifting and auto-scaling, all the dimensions of $dv \in DV$ have the same mean value and standard deviation. To avoid negative $dv_i$, it will be adjusted by subtracting the minimum $dv_i$ of all types of node (edge) pairs. Equation (1) cannot be evaluated if the standard deviation is 0, which indicates that the $dv_i$ of all the distance vectors $dv \in DV$ are the same, This actually indicates a bad choice of characteristic property, vector representation or distance calculator, since this dimension cannot help to distinguish the model elements (relations). *GenericDiff* ignores such dimensions when computing the distance between two composite vectors.

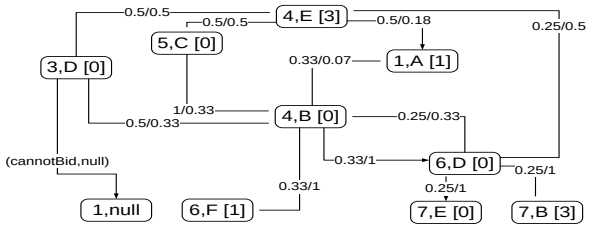


**Figure 4 The partial PairUpGraph of two state models**

Figure 4 presents a partial PairUpGraph of the TAGs of the Auctioneer and Bidder models. The initial distance value of state pairs (i.e., PairUpGraph nodes) is shown in the square bracket. We do not use normalized distance values here for the reason of clarity. The metrics on transition pairs (i.e., PairUpGraph edges) will be explained shortly in Section 3.5.

The construction of PairUpGraph is guided by a set of user-defined pairup feasibility predicates. There are two kinds of pairup feasibility predicates, respectively regarding the domain-specific types/properties and syntax of the models to be compared. The feasibility predicates define (but not limited to) the type compatibilities, minimum property similarities and topological constraints that a pair of graph nodes (edges) must satisfy so that they can be paired-up as matching candidates. These predicates are evaluated against a fact base (e.g., a relational database in current implementation of *GenericDiff*) of input model graphs $G_l(V_l,E_l)$ and $G_r(V_r,E_r)$ and PairUpGraph $PUG(V_{pu},E_{pu})$.

In our running example, we specify three type/property-based feasibility predicates as follows:

$$f(e_l, e_r) = (e_l \text{ is } S \;\&\; e_r \text{ is } R) \mid (e_l \text{ is } R \;\&\; e_r \text{ is } S)$$
$$f(n_l, n_r) = f_1(n_l, n_r) \wedge f_2(n_l, n_r)$$
$$f_1(n_l, n_r) = (n_l.kind == n_r.kind)$$
$$f_2(n_l, n_r) = d(n_l, n_r) < min\big(d(n_l, null), d(null, n_r)\big)$$

The first predicate $f(e_l, e_r)$ defines the type compatibility of transitions. Since we compare two interacting state models, a *Send* (*R*eceive) transition in one model can only be paired-up with a *R*eceive (*Send*) transition in the other model. The second predicate $f_1(n_l, n_r)$ defines that only the same kind of states can be paired-

up. The third predicate $f_2(n_l, n_r)$ defines that two states can be paired-up iff their distance value is less than the minimum distance value of the relevant states to null. For example, the state 2 of the Bidder model and the state B of Auctioneer model will not be paired-up, since the number of their *incoming* and *outgoing* transitions is "too" different. In Section 4.1.2, we will present some examples of syntax-based feasibility predicates (e.g., constraints on containment hierarchy) for comparing class models.

Given the initial PairUpGraph, *GenericDiff* can optionally create and append null node (edge) pairs to the PairUpGraph as defined in the following equation:

$$[n_l', null] \in V_{pu} \wedge [e_l, null]^{[n_l, n_r] \to [n_l', null]} \in E_{pu}$$
$$\leftrightarrow [n_l, n_r] \in V_{pu} \wedge e_l^{n_l \to n_l'} \in E_l \wedge \forall e_r^{n_r \to n_r'} \in E_r$$
$$\wedge \nexists \left([n_l', n_r'] \in V_{pu} \wedge [e_l, e_r]^{[n_l, n_r] \to [n_l', n_r']} \in E_{pu}\right)$$

Let $e_l$ be an edge in $G_l$ and let the source $n_l$ of $e_l$ be paired-up with a node $n_r$ in $G_r$, if $e_l$ has not been paired-up with any edge $e_r$ originating from $n_r$, then a null edge pair *($e_l$,null)* to a null node pair *($n_l$',null)* will be created and appended to the node pair *($n_l$,$n_r$)*. Similarly, the null node (edge) pair can be created for the target node of $e_l$ and for the edges in $G_r$. The null node (edge) pairs capture the discrepancies of the structural context of a pair of nodes. Take the state pair (3,D) as an example. The edge *cannotBid* from the state 3 to state 1 in the Bidder model has not been paired-up with any edges originating from the state D in the Auctioneer model, since state 1 cannot be paired-up with the state B and E (they are different *kinds* of states). Consequently, a null edge pair to a null node pair have been appended to the state pair (3,D). In contrast, the state pair (6,D) does not link to any null node (edge) pairs, since all the edges from (to) the state 6 have been paired-up with some edges from (to) the state D.

## 3.5 Distance propagation by random walk

Given a PairUpGraph *PUG*, *GenericDiff* performs a random walk [27] on *PUG*, which is an iterative process that propagates the distance value from node pairs to node pairs based on graph structure. Each iteration propagates the distance values one step forward along the edges, until the random walk stabilizes. Due to the presence of cycles in PairUpGraph, the distance values can thus be mutually reinforced.

A random walk on graph 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 step *t*. The probability distribution on all the nodes is represented by a vector $r(t) = [r_1(t), \dots, r_N(t)]$, *N* being the number of nodes in the graph. Each position in $r(t)$ is indexed for a graph node. The probabilities $r_n(t)$ are updated at each step as follows:

$$r_n(t+1) = \sum_{s \in G} jump(s,n) \times jump(s) \times r_s(t) \quad (2)$$
$$+ \sum_{s \in src(n)} follow(e^{s \to n}) \times follow(s) \times r_s(t)$$

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. Since they represent probabilities, their values must be normalized such that $jump(s) + follow(s) = 1$, $\sum_{s \in G} jump(s,n) = 1$, and $\sum_{s \in src(n)} follow(e^{s \to n}) = 1$.

By default, *GenericDiff* assumes a random walk on PairupGraph for which the action bias $jump(s)$ and $follow(s)$ are independent of the node pair *s*. Thus, *GenericDiff* takes as input a parameter$df \in (0,1)$ (0.85 by default) such that $follow(s) = df$ and $jump(s) = 1 - df$. $df$ defines the extent to which the random walk depends on the local attributes and on the graph topology. We define $jump(s,n)$ and $follow(e^{s \to n})$ as follows:

$$jump(s,n) = \frac{F_j(s,n)}{\sum_{z \in G} F_j(s,z)}$$
$$follow(e^{s \to n}) = \frac{F_f(e^{s \to n})}{\sum_{z \in trg(s)} F_j(e^{s \to z})}$$

where $F_j(s,n)$ and $F_f(e^{s \to n})$ represent the tendency functions of jumping from the node pair *s* to the node pair *n* or following an edge pair $e^{s \to n}$ from *s* to *n*. Using this definition, both $jump(s,n)$ and $follow(s,n)$ meet the normalization constraints required by the random walk model except for the nodes without outgoing edges (i.e., sink nodes). For these nodes, we set $follow(s_{sink}) = 0$ and $jump(s_{sink}) = 1$.

The tendency function $F_j(s,n)$ and $F_f(e^{s \to n})$ are usually defined according to domain-specific properties. By default, *GenericDiff* defines $F_j(s,n) = d(n)$ and $F_f(e^{s \to n}) = d(e^{s \to n}) + d(n)$, i.e., as a linear function of the distance value of relevant node and edge pairs. In this definition, the random walk will be more likely to jump to node pairs or to follow edges which link nodes having similar properties. We adopt this default definition of tendency functions in our running example. Figure 4 show the resulting $follow(e^{s \to n})$ probabilities on the PairUpGraph edges (transition pairs). The two numbers are the forward and reverse (see below) $follow(e^{s \to n})$ probabilities respectively. Note that $F_j(s,n)$ and $F_f(e^{s \to n})$ become constants if the model elements and relations do not have any characteristic properties. This results in $jump(s,n) = 1/N$, i.e., the target of a jump is selected using a uniform probability distribution over all the *N* node pairs in the PairUpGraph. Similarly, all the edges from node pair *s* have the same probability to be followed, i.e., $follow(e^{s \to n}) = 1/|trg(s)|$, where $|trg(s)|$ is the number of outgoing edges of *s*.

Equation (2) is recursive. A solution to the Equation (2) for a PairupGraph can be reached by power iteration method [32] to a fix point. We define the initial vector $r(0) = [d(1), \dots, d(N)]$, i.e., the initial distance value of all *N* nodes in PairUpGraph. *GenericDiff* keeps computing $r(t+1)$ until the Euclidean distance between $r(t+1)$ and $r(t)$ becomes less than a parameter $\epsilon > 0$, or it stops the computation after a user-defined maximum number of iterations. After the power iteration method terminates, the vector *r*is normalized by mean-shifting and auto-scaling.

**Table 1 The rank vector of state pairs**

| Rank | Pair | Rank | Pair | Rank | Pair |
|---|---|---|---|---|---|
| 1 | 1,A | 5 | 6,F | 10 | 7,D |
| 2 | 7,E | 6 | 4,B | … | … |
| 3 | 5,C | … | … | 15 | 3,F |
| 4 | 6,D | 9 | 3,D | … | … |

Given a PairUpGraph *PUG*, *GenericDiff* performs two random walks: one is on the *PUG* and the other on the $PUG_{reverse}$ obtained by reversing the edges of the original *PUG*. Let $r = [r_i]_{i=1..N}$ and $r_{reverse} = [r_{i\_reverse}]_{i=1..N}$ be the stable probability vectors obtained from the two random walks respectively. It was shown [21] that $r_i$ and $r_{i\_reverse}$, i.e., the probability of being at the $i^{th}$ node pair after lots of exploration of PairUpGraph exactly models the correspondence of that pair of nodes. That is, a pair of nodes with

high probability of being visited can be thought of as being "similar" to each other. *GenericDiff* computes a rank vector

$$R = \left[\sqrt{r_i^2 + r_{i\_reverse}^2}\right]_{i=1..N}$$

where $r_i$ and $r_{i\_reverse}$ are the $i^{th}$ element of $r$ and $r_{reverse}$ repsectively. Table 1 shows the rank vector $R$ (partial) for the running example. After the distance propagation, it becomes clear now that state pair (6,D) are more similar than state pair (3,D)

## 3.6 Bipartite matching

*GenericDiff* reduces the problem of *Maximum Common Subgraph* (MCS) into a bipartite graph matching, for which polynomial algorithms exist. Given two TAGs $G_l(V_l,E_l)$ and $G_r(V_r,E_r)$, the PairUpGraph $PUG(V_{pu},E_{pu})$ and the rank vector $R$, *GenericDiff* constructs a bipartite graph $BG(S,T,E,W)$ as follows:

$$S = \{n_l | n_l \in V_l \wedge [n_l, ?] \in V_{pu}\}, T = \{n_r | n_r \in V_r \wedge [?, n_r] \in V_{pu}\}$$
$$E = \{e^{n_l \to n_r} | [n_l, n_r] \in V_{pu}\}, W: E \to R$$

where $S$ and $T$ are two disjoint sets that consist of nodes of one of the two TAGs that have been paired-up with some nodes of the other TAG, $E$ contains edges that connect a node in $S$ to one in $T$ iff two nodes have been paired-up, $W$ is a weight function that maps an edge in $E$ to the correspondence measure of the corresponding node pair. *GenericDiff* solves the bipartite matching using two algorithms, i.e., Gale-Shapley (GS) algorithm [14] for finding 1-to-1 stable matching and Hospital Resident (HR) algorithm [19] for many-to-1 stable matching. Given the preference lists of $n_l \in S$ and $n_r \in T$, a bipartite matching is stable iff there are no two nodes $n_l$ and $n_r$ who prefer each other to their current partners. Note that both algorithms analyze the relative preferences of node pairs instead of their absolute correspondence measures to find an optimal matching.

At first glance, the formulation of MCS problem as bipartite matching seems like a bad idea, since it throws away all the important graph structure, until one recalls that the graph structure is really encoded in the correspondence measures of node pairs during the distance propagation process. However, due to the fact that the selection of each node pair is considered individually, there is nothing in bipartite matching formulation that ensures that the global structure among corresponding node pairs are obeyed. For example, GS algorithm reports that state pair (3,F) as a match. Unfortunately, state 3 and state F should not be paired-up at all if one knows that state pair (4,B) is a match, since the selection of (4,B) rules out the possibility of the candidate (4,C) being a match and (3,F) cannot be paired-up if (4,C) were not paired-up.

To address this issue, *GenericDiff* combines the above bipartite matching with a greedy, best-first search to eliminate the candidate pairs that violate the global structure of already selected pairs. Let $(n_l,n_r)$ be a selected node pair. Let $(n_i,n_j)$ be a candidate pair in the initial collection of impossible pairs $\{(n_l,?)\} \cup \{(?,n_r)\}$, i.e., all other node pairs with $n_l$ or $n_r$. *GenericDiff* detaches $(n_i,n_j)$ from its neighboring node pairs. For an affected neighboring node pair, if it has not yet been selected as a match and it becomes isolated after detachment, i.e., not incident to any edges, then this neighboring node pair is appended to the collection of impossible pairs. This process continues until the collection of impossible pairs is empty. And then, *GenericDiff* returns to the bipartite matching to select another node pair as a match. Using this strategy, state pair (3,F) will be marked as an impossible pair after the selection of state pair (4,B), which prevents it from being selected by *GenericDiff*.

The bipartite matching has also been used to determine the correspondence of the edges of matched node pairs. Given two matched node pairs $s(s_l,s_r)$ and $n(n_l,n_r)$, i.e., $s(s_l,s_r)$ and $n(n_l,n_r)$ in the stable matching, *GenericDiff* constructs a bipartite graph $BG(S,T,E,W)$ based on the edge pairs $e(e_l,e_r)$ from the node pair $s$ to node pair $n$, where $S = \{e_l\}$, $T = \{e_r\}$, $E = \{e^{e_l \to e_r} | [e_l, e_r] \in E_{pu}\}$ and $W: e^{e_l \to e_r} \to d(e_l, e_r)$, and then use GS algorithm to determine edge correspondences.

## 3.7 The output of GenericDiff

*GenericDiff* reports a symmetric difference between two model graphs $G_l(V_l,E_l)$ and $G_r(V_r,E_r)$, i.e., a set $M$ of matched model elements and relations and two sets $I_l$ and $I_r$ of unmatched model elements and relations. Each match in $M$ represents a pair of model elements or relations, one from each model, which are reported by *GenericDiff* as matching. The sets $I_l$ and $I_r$ consist of model elements and relations that are only present in $M_l$ or $M_r$, i.e., $I_l = (\{n_l | n_l \in V_l \wedge \nexists [n_l, ?] \in M\}, \{e_l | e_l \in E_l \wedge \nexists [e_l, ?] \in M\}$ and $I_r = (\{n_r | n_r \in V_r \wedge \nexists [?, n_r] \in M\}, \{e_r | e_r \in E_r \wedge \nexists [?, e_r] \in M\})$.

In our running example, the match set $M$ contains five pairs of matched states [A,1], [B,4], [C,5], [D,6] and [E,7]; it also contains seven pairs of matched transitions, such as, $[-itemSold^{B \to A}, +gavel^{4 \to 1}]$ and $[+bid^{B \to D}, -newBid^{4 \to 6}]$. Further examining the messages of matched transitions reveals the incompatible messages between two interacting models. Furthermore, the unmatched set $I_l$ contains one unmatched state F and three unmatched transitions in the Auctioneer model (highlighted in red); $I_r$ contains two unmatched states (state 2 and 3) and five unmatched transitions from the Bidder model (highlighted in blue). These unmatched states and transitions reveal the behavior inconsistencies between the Auctioneer and Bidder protocols.

## 3.8 The complexity of *GenericDiff*

*GenericDiff* offers polynomial time complexity. The parsing of an input model into a typed attributed graph scans sequentially the model elements and relations and encode their properties, and thus its complexity is $O(|V|+|E|)$ where $|V|$ and $|E|$ are the number of model elements and relations respectively. The construction of PairUpGraph examines the pairup feasibility of edges of two models to be compared and its worse case complexity is $O(|E_l| \times |E_r|)$. The complexity of random walk on PairUpGraph is proportional to the number of edges in the PairupGraph, which is in turn proportional to the product of edge numbers of two models. Thus, the complexity of random walk is also $O(|E_l| \times |E_r|)$. Finally, the complexity of stable marriage [14] and hospital resident algorithm [19] is $O(|V_l| \times |V_r|)$.

# 4. Evaluation

In this section, we present three applications of *GenericDiff*: compare UML class models of object-oriented software, compare the operational semantic models (LTSs) of CSP# specifications, and compare the PDGs of code clones. We focus on two research questions. First, can the domain-specific types/properties and syntax be effectively encoded in the specification of domain-specific inputs to *GenericDiff*? Second, is *GenericDiff* comparably accurate to domain-specific differencing algorithms?

## 4.1 Understanding software design evolution

Recognizing the changes that a system has gone through its lifecycle is essential to understanding how and why a system has reached its current state. We have applied *GenericDiff* to identify the renamings and moves of program entities as object-oriented

software evolves. Renaming and move are two types of elementary changes in the evolution of object-oriented software. Recognizing them is the goal of several domain-specific differencing algorithms, such as UMLDiff [47] for comparing UML class models, since it constitutes an essential prerequisite for the accurate analysis of software design and its evolution [24,31,47]. In this application, we applied *GenericDiff* to the empirical data used in [47] to evaluate UMLDiff. The data set consists of the reverse-engineered UML 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. This allows us to comparatively evaluate the accuracy of *GenericDiff* and *UMLDiff* in recognizing renamed and moved program entities.

### *4.1.1 Domain-specific types and properties*

Table 2 summarizes the metaclasses and metarelations for the reverse-engineered class models of Java systems that UMLDiff compares [47]. According to this metamodel, when parsing the reverse-engineered class model, *GenericDiff* builds a TAG, consisting of graph nodes whose type attribute corresponds to one of the nine metaclasses of model elements, and consisting of graph edges whose type attribute corresponds to one of the 13 metarelations and metaassociations of relations between model elements. Note that we consider stereotypes as distinct types in the specification of domain-specific types for *GenericDiff*, since different stereotypes represent distinct semantics of model elements and relations. For example, Usage$_{<<call>>}$ represents the invocation relation between two operations, while Usage$_{<<read>>}$ represents that an operation accesses the content of a property.

**Table 2 The metamodel for reverse-engineered class models**

| | |
|---|---|
| Metaclass | Subsystem (Subsys), Package (Pkg), Class, Interface, DataType, Property, Operation, Operation$_{<<constructor>>}$, Parameter |
| Metarelation | Generalization, Abstraction$_{<<realize>>}$, Usage$_{<<call>>/<<instantiate>>/<<read>>/<<write>>/<<throw>>}$ |
| Metaassociation | ElementOwnership, DeclaredParameter, FeatureType, ParameterType, DeclaredException, CatchException |

**Table 3 The composite vector attribute of model elements**

| | |
|---|---|
| Subsys/Pkg | $[\lvert ownedElement \rvert]$ |
| Class/Inteface | $[\lvert ownedElement \rvert], [\lvert InUsage \rvert, \lvert OutUsage \rvert]$ |
| Property | $[\lvert U_{\ll read \gg} \rvert, \lvert U_{\ll write \gg} \rvert]$ |
| Operation | $[\lvert DeclaredParameter \rvert], [\lvert U_{\ll call \gg} \rvert]$ $[\lvert U_{\ll read \gg} \rvert, \lvert U_{\ll write \gg} \rvert, \lvert U_{\ll call \gg} \rvert, \lvert U_{\ll throw \gg} \rvert]$ |

UMLDiff compares the name property and neighborhood (i.e., related elements) of two elements; it computes two similarity metrics, i.e., lexical and structure similarity for identifying corresponding elements in the two versions of the class model. The lexical similarity refers to the string similarity between the names of two model elements. UMLDiff splits names into a sequence of words, using dots, dashes, underscores and case switching as delimiters and computes the longest common subsequence (LCS) [45] for measuring the similarity between two names. Accordingly, to adapt *GenericDiff*, we specify the name as a domain-specific property of model element. We use word sequence as the representation of names and select the longest common subsequence to measure the similarity between names.

The structure similarity of two elements measures the overlap between the set of elements to which the two elements are related, according to a given relation type. For example, the structure similarity of two operations is determined by the parameters they declare, their incoming usage dependencies (the operations that call them), and their outgoing usage dependencies (the properties they read and write, the operation they call, the classes/interfaces they instantiate and the exceptions they throw). Accordingly, we define a composite numeric vector for approximating the neighborhood of different types of elements (see Table 3). The numeric vector summarizes the number of the related elements of a given relation type. We select Manhattan distance to measure the differences between the neighborhoods of two elements.

### *4.1.2 Pairup feasibility predicates*

The UML specification [54] guarantees that all model elements can be visited by traversing the containment hierarchy (i.e., the spanning tree induced by *ElementOwnership* relations) of a class model and the children of their containing parent are unique in terms of their names. UMLDiff exploits this domain-specific heuristics to prune matching candidates: (H1) it considers as matches the elements with the same Fully-Qualified Names (FQNs); (H2) it considers as matches the same-name elements contained in a pair of renamed or moved elements; (H3) it considers as renaming candidates only the unmatched elements within the context of a pair of mapped containing elements; (H4) it considers as move candidates only the unmatched same-name elements. We define three pairup feasibility predicates for *GenericDiff* to prune the matching candidates accordingly:

$$f(n_l, n_r) = f_1(n_l, n_r)\ \&\ \big(f_2(n_l, n_r) \mid f_3(n_l, n_r)\big)$$

$$f_1(n_l, n_r) = s_1(n_l, n_r) \mid s_2(n_l, n_r)$$

$$f_2(n_l, n_r) = s_3(n_l, n_r)\ \&\ \Big(s_4(n_l, n_r) \mid \big(s_5(n_l, n_r)\ \&\ s_6(n_l, n_r)\big)\Big)$$

$$f_3(n_l, n_r) = !\, s_3(n_l, n_r)\ \&\ \big(s_4(n_l, n_r)\ \&\ s_5(n_l, n_r)\ \&\ s_6(n_l, n_r)\big)$$

$$s_1(n_l, n_r) = (n_l.FQN == n_r.FQN)$$

$$s_2(n_l, n_r) = (\nexists n_l' \in V_l, n_l'.FQN == n_r.FQN)\ \&\ (\nexists n_r' \in V_r, n_l.FQN == n_r'.FQN)$$

$$s_3(n_l, n_r) = [n_l.parent, n_r.parent] \in V_{pu}$$

$$s_4(n_l, n_r) = (n_l.name == n_r.name)$$

$$s_5(n_l, n_r) = \nexists n_l' \in V_l, [n_l'.parent, n_r.parent] \in V_{pu}\ \&\ n_l'.name == n_r.name$$

$$s_6(n_l, n_r) = \nexists n_r' \in V_r, [n_l.parent, n_r'.parent] \in V_{pu}\ \&\ n_l.name == n_r'.name$$

The predicate $f_1$ enacts the UMLDiff heuristics H1. An element in one model can only be paired-up with the element of the same fully-qualified name in the other model if there is such a counterpart. The predicate $f_2$ enacts H2 and H3. Given two elements $n_l$ and $n_r$, if their containing elements are paired-up ($s_3$), the two elements can be paired-up if they have the same-name ($s_4$) or if there are no other elements $n_l'$ ($n_r'$) whose containing element (*parent*) is paired-up with the containing elements of $n_r$ ($n_l$) and whose name is the same as that of $n_r$ ($n_l$) ($s_5$ and $s_6$). The predicate $f_3$ enacts H4. Given two elements $n_l$ and $n_r$, if their containing elements are not paired-up ($s_3$), the two elements can be paired-up if they have the same-name ($s_4$) and if there are no other elements $n_l'$ ($n_r'$) whose containing element is paired-up with the containing elements of $n_r$ ($n_l$) and whose name is the same as that of $n_r$ ($n_l$).

In addition to the above four heuristics, UMLDiff considers two elements as renaming or move candidates only if their similarity

metric is above the user-specified renaming or move thresholds. Accordingly, we define another predicate as follows:

$$f_4(n_l, n_r) = (|\{n_l'\}| > 0 \mid |\{n_r'\}| > 0)\ \&\ \frac{\min(|\{n_l''\}|, |\{n_r''\}|)}{|\{n_l'\}| + |\{n_r'\}|} > T$$

where $n_l' \in neb(n_l)$ , $n_r' \in neb(n_r)$ , $[n_l'', n_r''] \in neb([n_l, n_r])$ represents the neighboring elements or element pairs of $n_l$, $n_r$ and $[n_l, n_r]$ in the input graphs $G_l$, $G_r$ and PairUpGraph *PUG*, respectively, and *T* is a user-specified threshold. We set *T* at 0.3, which is the threshold used in [47] to evaluate UMLDiff. The predicate $f_4$ defines that two elements can only be paired-up if there are "enough" related elements being paired-up. Note that according to $f_4$, an element cannot be paired-up with an "isolated" element, i.e., the elements with no neighboring elements.

### *4.1.3 Other domain-specific inputs*

UMLDiff shows that the similarity between two elements can be inferred by considering the similarity of their neighbors [47]. Thus, in the application of *GenericDiff* for comparing UML class models, we use the default *GenericDiff* random walk tendency functions $F_j(s,n) = d(n)$ and $F_f(e^{s\to n}) = d(e^{s\to n}) + d(n)$ , which defines $F_j(s,n)$ and $F_f(e^{s\to n})$ as linear functions of the distance value of relevant node pair and edge pair.

In the evolution of object-oriented software, refactorings often results in many-to-1 correspondences between model elements. For example, *extract superclass* moves a few methods from subclasses to a new superclass. Thus, we configure *GenericDiff* to use HR algorithm [19] for finding many-to1 matching.

### *4.1.4 Results*

The matching process of UMLDiff is hardcoded with the specific element/relation types, properties and heuristics that are effective for the comparison of UML class models. In this section, we defined these domain-specific types, properties and heuristics as the specification of domain-specific inputs to *GenericDiff*, separated from its generic matching process. Overall, *GenericDiff* achieves the comparable accuracy to UMLDiff. The GenericDiff's precision and recall in recognizing renamed and moved program entities is 92% and 88% for JFreeChart and 97% and 98% for HtmlUnit. The precision and recall of UMLDiff is 91% and 93% for JFreeChart and 95% and 98% for HtmlUnit [47].

The recall of *GenericDiff* in JFreeChart case study is slightly worse than that of UMLDiff. This is because the reverse-engineered class models contain certain amount of elements with no related elements [47,48], such as the methods with no incoming and outgoing usage dependencies or the empty interfaces. The composite vector attributes of these elements are all zero vectors. Furthermore, such elements when paired-up become the isolated nodes in PairUpGraph; they cannot effectively participate in the distance propagation process. Thus, we exclude them from the PairUpGraph (see $f_4$). Since these elements do not use or are used by other elements, ignoring them does not pose a significant threat to the quality of *GenericDiff*'s results.

## 4.2 Debugging evolving system behaviors

The Labeled Transition System (LTS) [7] provides a generic operational semantic model for analyzing and verifying the behavior of computer programs. Program behaviors (generated from the operational semantics) evolve as the program evolves. Even when a program remains unchanged, its LTS model explored by a model checker or analyzer may still change due to the application of different model optimization techniques [7]. Pinpointing the differences in the evolving LTSs of a program can lead to effective analysis of program faults and the behavioral change patterns of the program. In this application, we have applied *GenericDiff* to compare the evolving LTSs of CSP# [42] programs.

### *4.2.1 Syntax and operational semantics of CSP#*

CSP# (Communication Sequential Program #) [42] is a specification language for modeling and verifying the behavior of concurrent programs. A CSP# program consists of several process definitions, in the form of $P(x_1, x_2 \ldots, x_n) = ProcessExp$, where *P* is the process name, $x_1, x_2 \ldots, x_n$ is an optional list of process parameters and *ProcessExp* is a process expression. The process expression defines the computational logic of the process. Figure 5 presents the partial BNF description of CSP# process expression. CSP# supports various types of process constructs, including primitives, event prefixing, channel communication, hiding, and various process compositions. The CSP# parser parses a CSP# program into a configuration graph *CG(V,E)*, i.e., the internal syntactic model of a CSP# program. A *CG(V,E)* is a rooted directed graph, where the vertex set *V* contains the processes defined in the program and the edge set *E* contains the composition relations between the processes.

| | |
|---|---|
| P = Stop \| Skip | (primitives) |
| \| e.*x*{*prog*} → P | (event prefixing) |
| \| ch!*x* → P \| ch?*x* → P | (channel output/input) |
| \| P \ X | (hiding) |
| \| P[] Q | (choice operators) |
| \| if *b* {P} else {Q} | (conditional choice) |
| \| P; Q \| P \|\| Q \| P\|\|\| Q \| P Δ Q | (compositions) |
| \| *ref*(Q) | (process reference) |

**Figure 5 The BNF description of CSP# process expression**

The structural operational semantics of CSP# language [42] defines the behavior of a CSP# program in terms of a set of transition rules. These rules translate the syntactic model of a CSP# program into an LTS. An LTS is a 3-tuple (*S*, *init*, →), which consists of a set *S* of global states, the initial states *init*∈*S*, and a set of labeled transition relations →. A state is composed of two components (*V*, *P*) where *V* is a valuation function mapping a variable name (or a channel name) to its value (or a sequence of items in the buffer) and *P* is the current process expression. A transition is a labeled directed relation from a source state to a target state. The transition label represents the engaged event, which has a name and an ordered list (possible empty) of parameter expressions.

### *4.2.2 Comparing configuration graphs*

The process definitions may change as the CSP# program evolves. As they are a key factor to determine the state similarity in the corresponding LTSs, we first apply *GenericDiff* to compare the configuration graphs of two versions of a CSP# program in order to determine the corresponding process definitions in two versions of the program. *GenericDiff* parses a configuration graph into a TAG, consisting of graph nodes whose type attribute represents the type of the corresponding process defined in the program, and consisting of graph edges whose type attribute represents the composition relation between the processes.

We specify several characteristic properties for discriminating processes. First, we encode the name of event prefixing (*e*) and channel output/input (*ch*) process in a char sequence and use LCS [45] to measure the similarity between two names. Second, we encode the valuation of parameter expression *x* of event prefixing and channel output/input process in a numeric vector and use

Manhattan distance to measure the similarity between two parameter expressions. Third, we encode the boolean expression *b* of the conditional choice process and the sequential program *prog* attached to the event prefixing process in a char sequence and use LCS [45] to measure the similarity between two boolean expressions or sequential programs. Finally, we define a numeric vector for summarizing the number of composed processes of choice, parallel and interleave process. We use Manhattan distance to measure the similarity between the numbers of composed processes.

We define two pairup feasibility predicates for the comparison of configuration graphs. First, each type of CSP# process construct defines a distinct system behavior. It makes no sense to pairup process constructs of different types. Thus, we define that, given the graph nodes $n_l$ and $n_r$ of two processes, they can be paired-up iff $n_l.type == n_r.type$, i.e., their type attributes are the same. Second, we do not want two very different processes to be paired-up. Thus, we define that two graph nodes can be paired-up iff the distance between the two nodes is less than the minimum distance value of the node to null, i.e., $d(n_l, n_r) < min\big(d(n_l, null), d(null, n_r)\big)$.

As the graph edges of configuration graph represent the composition relations between processes, the similar process tend to be composed of other similar processes. Thus, we use the default random walk tendency function of GenericDiff framework, which are defined as linear functions of the distance value of relevant node and edge pair. Finally, we instruct GenericDiff to select a one-to-one matching between processes.

### 4.2.3 Comparing labeled transition systems

*GenericDiff* parses an input LTS into a TAG, consisting of graph nodes whose type attribute is state and graph edges whose type attribute is transition.

We specify two characteristic properties for discriminating state nodes. First, we encode the valuation *V* of global variables and channels at a state in a composite numeric vector, consisting of a numeric vector for the variables of primitive data types (integer and boolean) and a numeric vector for each variable of array or structured data type. Note that the definition of this composite numeric vector may be recursive, since the array and structure data type may contain other array and/or structured data types. Given the composite vector attributes of two state nodes, we use Manhattan distance to measure the similarity between the valuation of global variables and channels at two states. This distance computation may also be recursive.

Second, the process expression *P* at a state is another discriminating property of state nodes. However, comparing the process expression literally cannot express the similarity of the behaviors of states. In this application, we approximate the process expression *P* at state *S* with a set of active processes, which consists of the event prefixing and channel output/input processes obtained from *P* according to the structural operational semantics of CSP#. These active processes define the behavior of the state *S*, i.e., performing these processes advances the LTS one step further from the state *S*. Give the sets of active processes of two state nodes, we use Jaccard coefficient, a commonly used metric for comparing two sets of elements to measure the behavioral similarity of two states. When comparing the LTSs of two versions of a CSP# program, the process definitions may change as the program evolves. In this case, we rely on the matching results of comparing the corresponding configuration graphs to determine the correspondences between active processes.

The transition label (i.e., engaged event) is a discriminating property of the edges of the LTS graph. We encode the name of engaged event in a char sequence and use LCS [45] to measure the similarity between two event names. Furthermore, we encode the valuation of parameter expressions (if any) of engaged event in a numeric vector and use Manhattan distance to measure the similarity between the parameter expressions of two events.

We define one pairup feasibility predicate for the comparison of LTSs. We do not want two very different states to be paired-up. Thus, we define that $d(n_l, n_r) < min\big(d(n_l, null), d(null, n_r)\big)$, i.e., two state nodes $n_l$ and $n_r$ (or transition edges) can be paired-up iff the distance between the two nodes (or edges) is less than the minimum distance value of the node (or edge) to null.

Our formulation of the LTS similarity is a quantitative analogue of exact bisimilarity [29] in that similar states are linked to other similar states by similar transitions. Thus, we use the default random walk tendency function of GenericDiff framework, which are defined as linear functions of the distance value of relevant state and transition pair. Finally, we instruct GenericDiff to select a one-to-one matching between states and transitions.

### 4.2.4 Results

We applied *GenericDiff* to compare the LTSs of a correct and a faulty version of a concurrent stack program written in CSP#. The faulty version violates the linearizibility [17] of the concurrent stack due to the decrease of atomicity level of two conditional choices. The two LTSs contain 438 states and 1120 transitions and 1102 states and 2642 transitions, respectively. The precision and recall of *GenericDiff* in comparing these two LTSs is 95% and 95%. Inspecting the differences between the two LTSs identifies four types of incorrect interactions between two processes that result in the violation of the linearizibility of concurrent stack.

The process counter abstraction and the cutoff number is a common state abstraction technique for analyzing parameterized systems [7]. We applied *GenericDiff* to compare the 20 LTSs ($lts_i$ and $lts_{i+1}$, obtained by setting the cutoff number to 1..20) of a parameterized readers-writer-lock CSP# program. We also applied *GenericDiff* to compare the 9 LTSs of a parameterized Java meta-lock CSP# program. In both studies, the precision and recall of *GenericDiff* in comparing these LTSs is 100% and 100%. The differences reported by *GenericDiff* reveal the behavioral change patterns of parameterized systems as the cutoff number increases.

Partial order reduction [7] exploits the commutativity of concurrently executed transitions to reduce the state space to be explored by a model checking algorithm. We applied *GenericDiff* to compare the two LTSs of a dining philosophers CSP# program, explored with and without partial order reduction. GenericDiff perfectly (100% precision and 100% recall) isolates the states and transitions that have not been explored when the partial order reduction is in place. This helps the developers better understand the impact of partial order reduction technique.

There have been two domain-specific algorithms [30,41] for comparing state-machine-like models. Sokolsky et al. [41] compute the overall similarity of the LTSs of viruses to classify them into families. Nejati et al. [30] find and merge the differences between StateCharts requirement specifications. The random walk process of GenericDiff is analogue to the Markov decision process used in these two algorithms. But, as the objective and the output of our work and theirs are different, three approaches are not directly comparable. In terms of precisions and recalls, *GenericDiff* performs as well as these two domain-specific algorithms.

## 4.3 Semantic differencing of software clones

Clone detection [1,3,13,25] provides a scalable and efficient way to detect similar code fragments. But it offers little explanation about the characteristics of clones, such as how clones are different if they are not identical. Understanding these characteristics is crucial during post-detection clone analysis and program maintenance that affect clones. In this application, we complement clone detection with program differencing for the purpose of characterizing the differences of clones. We capture semantic information of clones from Program Dependence Graphs (PDGs) that encode data and control dependencies between program statements. We then adapt GenericDiff to compute the semantic differences of clones in terms of the structural differences and differential properties between their PDGs.

### *4.3.1 Clone detection*

In this application, we assume there is a clone detection method that can detect similar methods $m_1$ and $m_2$ (i.e., cloned methods) that contain one or more similar code fragments according to certain similarity measurement. We use CloneMiner [2] for the detection of code clones. CloneMiner finds simple clones (i.e., similar fragments of contiguous codes) first, using Repeated Tokens Finder, a token-based simple clone detection technique [1]. Then, it mines simple clones with frequent itemset mining [16] to detect structural clones across large program units, such as cloned methods that contain one or more simple clones.

### *4.3.2 Program dependence graph*

We adopt intra-method PDG [11] to capture semantic information of clones. A PDG is an intermediate program model that encodes both the data and control dependences between program statements. Given a cloned method *m*, we use Wala [55], a static analysis library for Java bytecode, to generate the PDG of the method *m*. Wala represents program statements in a SSA-based register transfer language.

The nodes of a Wala-PDG consist of the SSA statements constructed from the source code. Wala supports 23 types of SSA statements. We categorize them into three groups: operation statements, branch statements, and parameter/constant statements. A SSA statement has the following properties: a collection of symbols that it *uses*, at most one *result* symbol that it defines, a collection of *incoming* dependences and/or a collection of *outgoing* dependences. A symbol is a unique id representing a variable or value. Different concrete types of SSA statements can have different sets of additional properties.

The edges of a PDG represent the control and data dependences between SSA statements. Given two SSA statements, $s_1$ and $s_2$, the data dependence from $s_1$ to $s_2$ means that the value produced at $s_1$ may be used at $s_2$. A control dependence from $s_1$ to $s_2$ means that the choices of executing $s_2$ depends on the test the evaluation of $s_1$. A control dependence may have an optional *label*, representing the corresponding choice.

### *4.3.3 Computing semantic differences of clones*

*GenericDiff* parses an input PDG into a TAG, consisting of graph nodes whose type attribute represents the type of the corresponding SSA statements [10], and consisting of graph edges whose type attribute represents either control or data dependence.

We specify the following properties of SSA statements and dependences for discriminating graph nodes and edges. First, we ignore the *result* and *uses* symbols of program statements, since they are not stable across PDGs. Second, we encode enumeration properties of an SSA statement or dependence in an enumeration-literal vector associated with the corresponding graph nodes and edges. Such enumeration properties include: the *label* of control dependences, the *operator code* of unary operation (negate), binary operation (e.g., add, minus, multiply), and compare and branch (e.g., >, <, !=) statements, and the *isEnter* of synchronization statements. We also encode the *constant* value of constant statements in a literal vector. Since we are interested in whether two enumeration or constant values are different, hamming distance is used to measure the similarity between two literal vectors.

Third, we ignore the *signature* of methods being invoked in *INVOKE* statements and the *signature* of fields being accessed in *FGET* (field read) and *FPUT* (field write) statements, since they can be different across cloned methods. However, we specify the return type of methods being invoked and the type of field being accessed as a characteristic property of *INVOKE* and *FGET/FPUT* statements. The underlying intuition is that the methods and fields that have different types may play different roles in cloned methods. We encode the return type of methods being invoked and the type of fields being accessed in a typename-literal vector. We also encode the relevant type properties of other kinds of SSA statements to discriminate the corresponding graph nodes, such as the *element type* of *ARRAYLOAD* and *ARRAYSTORE* statements, the *type* being checked in *INSTANCEOF* statements, and the *type* being instantiated in *NEW* statements. Similar to the comparison of enumeration-literal vectors, we use hamming distance to measure the similarity between two typename-literal vectors.

Forth, we specify a numeric vector of two elements to summarize the number of *incoming* and *outgoing* dependences of a SSA statement. We also specify a numeric vector to encode the number of *branches* of *SWITCH* statements, the *array dimension* of *NEW* statements, and the number of *assignments* of *PHI* statements. As these numeric vectors represent the size of a collection, we use Manhattan distance to measure their similarity.

We define one pairup feasibility predicate for the comparison of PDGs. Since different types of program statements (dependences) define distinct semantics [36], we define that two nodes (edges) can be paired-up as matching candidates iff their type attributes are the same.

We assume that similar SSA statements are related to other similar statements in two PDGs. Thus, we use the default random walk tendency functions provided by GenericDiff framework. Finally, we specify GenericDiff to select a one-to-one matching between program statements (dependences).

### *4.3.4 Results*

Given two PDGs of a pair of cloned methods ($m_1$,$m_2$) and the matching results by GenericDiff, we identify the following five types of characteristic differences of cloned methods by searching for certain patterns in the two PDGs and their differences:

1. *Differential properties* summarize all the pairs of matched program statements that have different properties.
2. *Additional branches* are the unmatched branch statements before or after a pair of matched operation statements.
3. *Partially matched branches* are the unmatched branch statements before or after a pair of matched branch statements.
4. *Additional operations* are the unmatched operation statements before or after a pair of matched statements.
5. *Unmatched operation pairs* are the pairs of unmatched same-type operation statements before or after a pair of matched statements.

We have implemented a tool Clone Differentiator that supports the PDG differencing of code clones with GenericDiff. We have evaluated Clone Differentiator on three Java systems, Java IO library, Berkeley Database and Eclipse Plugins.

By focusing on areas that are known to be highly similar, the accuracy of GenericDiff in comparing the PDGs of clones is good. We manually inspected the PDG comparison results of randomly-selected 10% of all the analyzed clone pairs. The precision (i.e., the percentage of the correctly reported matches) and the recall (i.e., the percentage of matched reported) of GenericDiff is around 94% and 96%, respectively.

Furthermore, these evaluations found PDG differencing of clones useful in the following three scenarios:

*Suggesting appropriate refactoring actions.* Code cloning can result in unused, dead code in the system that hinders program comprehension and maintainability [12]. Detecting such code fragments and analyzing how different they are help developers to decide what action to take. For example, in our case studies, Clone Differentiator can assist in distinguishing identical dead code, methods that are "part of" another, parallel inheritance hierarchies, and deviations of design patterns.

*Consistent management of clones.* Clones pose additional problems if they do not evolve synchronously [22,26]. Detecting the differences of clones raises the awareness of their inconsistencies so that they can be graceful handled. For example, in Java IO library, Clone Differentiator revealed three inconsistent programming styles of validating the input parameters, handling null exceptions and using synchronization. We also found the inconsistencies of clones be indicative of potential bugs.

*Identifying variations of a common solution*. Reusing and adapting a common solution in different contexts can prevent errors by reusing trusted solutions. This often results in code clones with variations, depending on the context. In our case, we found that analyzing the PDG differences of clones helps developers better understand the commonalities and variations of similar solutions in different contexts.

## 5. Conclusion and future work

In this paper, we presented *GenericDiff*, a general framework for software model comparison. It exhibits several advantages over the current state of the art. Because it separates the specification of domain-specific inputs from the generic matching process, it is easy to adapt *GenericDiff* in a new application domain. Because it encodes domain-specific properties and syntax in two generic data structures (i.e., TAG and PairUpGraph), the domain-specific properties and syntax can be uniformly explored in the generic matching process. Because it leverages the useful techniques (i.e., random walk on graph and bipartite matching) developed in pattern mining and graph matching, it is capable of producing an accurate comparison report for diverse types of models.

We have implemented *GenericDiff* and adapted it in three applications for comparing UML class models, labeled transition systems, program dependence graphs and product feature models. To the best of our knowledge, there has been no other algorithm that has such broad coverage and can still produce accurate and useful comparison report.

Our plans for future work include applying *GenericDiff* to compare heterogeneous models, i.e., a mixture of different types of models. We also plan to extend *GenericDiff* to other application areas, such as the protocol adaptation for service integration and debugging programs with evolving requirements.

## References


1 H.A. Basit, S. Puglisi, W. Smyth, A. Turpin and S. Jarzabek. Efficient token-based clone detection with flexible tokenization. *ESEC/FSE* 2007, pp. 513-516.

2 H.A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones. *TSE*, 35(4): 497-514, 2009.

3 I.D. Baxter, A. Yahin, L. Marcelo, M. Sant'Anna and L. Bier.: Clone detection using abstract syntax trees. *ICSM* 1998, pp. 368-377.

4 C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM* 16:575-577, 1973.

5 C. Canal, P. Poizat and G. Salaun. Model-based adaptation of behavioral mismatching components. *TSE 34(4)*, pp. 546-563, 2008.

6 M. Christodorescu, S. Jha and C. Kruegel. Mining specifications of malicious behavior. *FSE*, pp. 5-14, 2007.

7 E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. The MIT Press, 1999.

8 D. Conte, P. Foggia, C. Sansone and M. Vento. Thirty years of graph matching in pattern recognition. *Journal of Pattern Recognition and Artificial Intelligence*, pp. 265-298, 2004.

9 L.P. Cordella et al. Subgraph transformations for inexact matching of attributed relational graphs. *Computing Suppl*, 12:43-52, 1998.

10 R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13(4): 451-490.

11 J. Ferrante, K.J. Ottenstein and J.D. Warren. The Program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319-349, 1987.

12 M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

13 M. Gabel, L. Jiang and Z. Su. Scalable detection of semantic clones. *ICSE* 2008, pp. 321-330.

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

15 M. Gori and M. Maggini. Exact and appropriate graph matching using random walks.

16 G. Grahne and J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. *Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003.

17 M.P. Herlihy and J.M. Wing. Linearizibility: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12(3):463-492, 1990.

18 S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *PLDI*,pp. 234-246, 1990.

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

20 D. Jackson and D.A. Ladd. Semantic diff: A tool for summarizing the effect of modifications. *ICSM*, pp. 243-252, 1994.

21 G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. *KDD*, pp. 538-543, 2002.s

22 C. Kapser and M.W. Godfrey. "Cloning Considered Harmful" Considered Harmful. *WCRE* 2006, pp. 19-28.

23 K.C. Kang, J. Lee and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58-65, 2002.

24 M. Kim, D. Notkin and D. Grossman. Automatic inference of structural changes for matching across program versions. *ICSE*, pp. 333-343, 2007.

25 R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *SAS* 2001, pp. 40-56.
26 J. Krinke. A Study of consistent and inconsistent changes to code clones. *WCRE* 2007, pp. 170-178.
27 L. Lovasz. Random walks on graphs: A survey. 1993.
28 S. Melnik, H.C. Molina, E. Rahm, Similarity flooding: A versatile graph matching algorithm and its application to schema matching. *ICDE*, pp. 177, 2002.
29 R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
30 S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook and P. Zave. Matching and merging startcharts specifications. *ICSE*, pp. 54-64, 2007.
31 D. Ohst, M. Welle, U. Kelter. Difference tools for analysis and design documents. *ICSM*, pp. 13-22, 2003.
32 L. Page et al. The PageRank citation ranking: Brining order to the web. Technical Report, Stanford Info Lab, 1999.
33 S. Person, M.B. Dwuer and C.S. Pasareanu. Differential symbolic execution. *ESEC-FSE*, pp. 226-237, 2008.
34 N.H. Pham, H.A. Nguyen, T.t. Nguyen, J.M. Kofahi and T.N. Nguyen Complete and accurate clone detection in graph-based models. *ICSE*, pp. 276-286, 2009.
35 D. Qi, A. Roychoudhury, Z. Liang and K. Vaswani. Darwin: an approach for debugging evolving programs. *ESEC-FSE*, pp. 33-42, 2009.
36 G. Ramalingam and T. Reps. Semantics of program representation graphs, Technical Report, University of Wisconsin Madison, 1989.
37 K. Riesen, S. Fankhauser, H. Bunke and P. Dickison. Efficient suboptimal graph isomorphism. *Graph-Based Representations in Pattern Recognition*, pp. 124-133, 2009.
38 G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.
39 R. Santelices, J.A. Jones, Y. Yu and M.J. Yarrold. Lightweight fault-localization using multiple coverage types. *ICSE*, pp. 56-66, 2009.
40 A. Shokoufandeh and S.J. Dickinson. A unified framework for indexing and matching hierarchical shape structures. *Int. Workshop on Visual Form*, pp. 67-84, 2001.
41 O. Sokolsky, S. Kannan and I. Lee. Simulation-based graph similarity. *TACAS*, pp. 426-440, 2006.
42 J. Sun, Y. Liu, J.S. Dong and C.Q. Chen. Integrating specifications and programs for system specification and verification. *TASE*, pp. 127-135, 2009.
43 P.N. Tan, M. Steinbach and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.
44 C. Treude, S. Berlik, S. Wenzel and U. Kelter. Difference computation of large models. *ESEC-FSE*, pp. 295-304, 2007.
45 R. Wagner and M. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168-173, 1974.
46 M.L. Williams, R.C. Wilson and R. Hancock. Structural matching by discrete relaxation. *IEEE Trans. Patt. Anal. Mach. Intell.* 19:634-648, 1997
47 Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. *ASE*, pp. 54-65, 2005.
48 Z. Xing. Model comparison with GenericDiff. *ASE*, 2010.
49 Z. Xing. GenericDiff: A general framework for model comparison. Technical Report, NUS, 2010.
50 Y. Xue, Z. Xing and S. Jarzabek. Understanding feature evolution in a family of product variants. *WCRE*, 2010.
51 X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. *Int. Conf. on Data Mining*, pp. 721, 2002.
52 D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, pp. 292-333, 1997.
53 EMF: http://www.eclipse.org/emft, 2010.
54 OMG UML: http://www.uml.org, 2010.
55 WALA: http://wala.sourceforge.net/wiki/index.php/Main_Page, 2010.


# Appendix 1

Table 4 summarizes the basic data types and their corresponding atomic vector attributes that *GenericDiff* supports. *GenericDiff* uses a set of simple distance calculators to determine the similarity of the properties of model elements and relations, encoded in vector attributes.

**Table 4 Atomic vectors and standard distance calculators**

| Dada type | Representation | Distance calculator |
|---|---|---|
| Enumeration | Literal index vector | Hamming |
| Numeric type | Numeric vector | Manhattan/Euclidean |
| String | Literal index vector | Hamming |
| | Word set/bag/sequence | Jaccard/LCS |
| | TF/IDF vector | Cosine similarity |
| Collection | numeric vector | Manhattan |
| | Set/bag/Sequence | Jaccard/LCS |

A property of enumeration type or constant values can be represented in a literal index vector. Each literal or constant is mapped onto a vector position. The value at that position is set to 1 for a property of that literal value. Since we are only interested in whether two literal or constant values are different, hamming distance is used to measure the similarity of two literal index vectors. Given two literal index vector $v_l$, and $v_r$, their hamming distance is $|\{i|v_l[i] \neq v_r[i]\}|$, i.e., the number of vector elements that are different.

Several properties of numeric type can be represented in a vector of numeric values. Depending on the metric space of numeric values, either Manhattan or Euclidean distance can be used to measure the similarity of two numeric vectors. Given two numeric vector $v_l$, and $v_r$, their Manhattan distance is $\sum_{i=1}^{n}|v_l[i] - v_r[i]|$, i.e., the sum of the value differences of corresponding vector elements; their Euclidean distance is the length of the line segments $\overline{v_l v_r}$, i.e., $\sqrt{\sum_{i=1}^{n}(v_l[i] - v_r[i])^2}$.

A property of String type can also be represented in a literal index vector when the set of string values of this property is fixed. Each position in the vector is indexed for a string value. In this case, hamming distance is used to measure the similarity of two vectors of string values.

For a property of String type, *GenericDiff* can exploit the common convention to split a string value into a set or bag of word. A set of words consist of unique word elements, while a bag of words may contains duplicate words. The Jaccard coefficient [43] is often used to measure the similarity of two sets or bags. Given two word sets/bags $v_l$, and $v_r$, their Jaccard coefficient is $|v_l \cap v_r|/|v_l \cup v_r|$, i.e., the size of the intersection of two sets divided by the size of their union. *GenericDiff* can also represent a string value as a sequence of words. In this case, the longest common subsequence (LCS) [45] is used to measure the similarity of two sequences.

*GenericDiff* can further analyze the word sets to generate a Term-Frequency/Inverse-Document-Frequency (TF-IDF) vector [38] for each string value. It is a statistical measure used to evaluate how important a word is to a document in a corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. The cosine similarity is often used to determine the distance between two TF/IDF vectors. Given two TF-IDF vectors $v_l$, and $v_r$, their cosine similarity is

$$\sum_{i=1}^{n} v_l[i]v_r[i] / \left(\sqrt{\sum_{i=1}^{n} v_l[i]^2}\sqrt{\sum_{i=1}^{n} v_r[i]^2}\right)$$

i.e., the dot product of two vectors divided by the product of the Euclidean distance of two vectors.

Since property is multiplicity element [54], it may represent a collection of values. A collection can be represented in an atomic vector. For example, the size of a collection can be encoded in a numeric vector. Alternatively, the values of a collection can be represented in a set or bag, depending on whether the property is unique. Or the collection can be represented in a sequence of values, depending on whether the property is ordered. The corresponding distance calculators can be used to measure the similarity of two collections.