Approximate Clone Detection in Repositories of Business Process Models

Chathura C. Ekanayake[1], Marlon Dumas[2], Luciano García-Bañuelos[2], Marcello La Rosa[1], and Arthur H.M. ter Hofstede[1,3]

[1] Queensland University of Technology, Australia
{c.ekanayake, m.larosa, a.terhofstede}@qut.edu.au
[2] University of Tartu, Estonia
{marlon.dumas, luciano.garcia}@ut.ee
[3] Eindhoven University of Technology, The Netherlands

Abstract. Evidence exists that repositories of business process models used in industrial practice contain significant amounts of duplication. This duplication may stem from the fact that the repository describes variants of the same processes and/or because of copy/pasting activity throughout the lifetime of the repository. Previous work has put forward techniques for identifying duplicate fragments (clones) that can be refactored into shared subprocesses. However, these techniques are limited to finding exact clones. This paper analyzes the problem of approximate clone detection and puts forward two techniques for detecting clusters of approximate clones. Experiments show that the proposed techniques are able to accurately retrieve clusters of approximate clones that originate from copy/pasting followed by independent modifications to the copied fragments.

1 Introduction

Duplication is a widespread phenomenon in software and model repositories [7, 11]. Not surprisingly, significant amounts of duplication can also be found in repositories of business process models used in industrial practice – both in the form of exact duplicates (a.k.a. *exact clones*) [14] and pairs of similar fragments (*approximate clones*) [4].[1] Clones in process model repositories emerge for example as a result of copy/pasting activity, but also when multiple variants of a process co-exist and are described as separate models. For example, a large insurance company typically runs multiple claims handling processes for different types of claims or products. Naturally, these process variants share some commonalities, which manifest themselves in the form of clones.

Detecting clones in process model repositories allows analysts to identify opportunities for standardization and refactoring. For example, given that disbursing occurs in multiple variants of a claims handling process, process fragments corresponding to disbursing can potentially be standardized and encapsulated in a shared subprocess. In previous work, we proposed a technique for identifying exact clones that can be refactored into shared subprocesses [14]. However, when clones emerge as a result of copy-/pasting, it is likely that these clones will subsequently undergo independent changes and thereon can no longer be detected using exact clone detection methods.

[1] The term fragment is used to refer to a connected subgraph of a process model.

When designing approximate clone detection methods, a first step is to define what an approximate clone is. Generally, such a definition relies on a similarity or (equivalently) a distance metric. Previous work has shown that graph-edit distance is a suitable proxy for perceived process model dissimilarity [3]. Accordingly, we postulate that a necessary condition for two process model fragments to be approximate clones is that their graph-edit distance is below a user-defined threshold. However, three additional issues ought to be considered when defining a notion of approximate clone.

Firstly, it should be considered that any fragment g_1 is similar to any fragment g_2 such that g_2 contains g_1 or g_1 contains g_2, provided that the difference between g_1 and g_2 falls below the threshold. A definition that would consider two fragments as approximate clones merely because one contains the other would lead to many false positives – an issue that has been widely discussed in the field of code and model clone detection [11]. Secondly, given the goal to identify approximate clones for the sake of refactoring them into subprocesses and given that subprocesses are invoked according to a call-and-return semantics, it is necessary that the approximate clones we retrieve are Single-Entry, Single-Exit (SESE) fragments. Thirdly, we are not interested in *trivial* clones consisting of a single activity, since they do not represent an opportunity for subprocess extraction. These considerations lead to the following definition.

Definition 1. *Given a distance metric $Dist$ and a distance threshold τ, two non-trivial, SESE process model fragments g_1 and g_2 are approximate fragments – written $Approx(g_1, g_2)$ – iff $g_1 \not\subset g_2$, $g_2 \not\subset g_1$ and $Dist(g_1, g_2) \leq \tau$.*

Armed with this definition, one can retrieve large amounts of approximate clone pairs [4]. However, if the goal is to help analysts to identify opportunities for refactoring and standardization, retrieving all such pairs is of limited use. Instead, given the goal at hand, analysts need to identify sets of fragments C that can be standardized towards a single fragment with a bounded amount of changes on each fragment. Otherwise, some fragments would need to undergo changes during the standardization that would convert them into arbitrarily different fragments. In this respect, we envisage two alternative approaches to standardize a set of fragments:

- A set of fragments can be standardized by taking a given "medoid" fragment as a reference and standardizing all fragments towards this medoid.
- A set of fragments can be standardized by selecting any fragment in the group as a reference and standardizing all other fragments towards this reference fragment.

This leads to the following definition.

Definition 2. *A set of SESE process model fragments C is a* cluster of approximate clones *iff one of the following properties holds:*

1. $\exists g \in C\ \forall g' \in C : Approx(g, g')$. *In this case, g is called the* cluster medoid.
2. $\forall g, g' \in C : Approx(g, g')$.

The main contribution of this paper are two techniques (one per standardization approach) for identifying clusters of approximate clones. The proposed techniques are validated in a twofold manner. First, a descriptive analysis of approximate clone clusters

in two industrial repositories of process models is undertaken. Secondly, a synthetic experiment is conducted to evaluate the accuracy of the clustering techniques with respect to the task of retrieving clusters of clones that have emanated from a single original fragment via copy/pasting followed by independent changes to the duplicated fragments.

The rest of the paper is structured as follows. Section 2 introduces three techniques for process model parsing, exact clone detection and process model comparison used in this paper. Section 3 then presents the proposed approximate clone clustering techniques. Finally Section 5 discusses related work while Section 6 concludes the paper.

2 Preliminaries

This section introduces the three basic ingredients of the proposed technique: RPST, RPSDAG and process model similarity.

2.1 RPST

The RPST [15, 12] is a parsing technique that takes as input a process model and computes a tree representing a hierarchy of single-entry single-exit (SESE) fragments. Intuitively, a process model, represented as a directed graph, is partitioned into sets of edges such that the subgraph induced by each set of edges is a SESE fragment. SESE fragments are organized by subset inclusion to form a rooted tree, where siblings are associated to disjoint sets of edges. As the process graph is partitioned into set of edges, some nodes may be shared in several SESE fragments. The RPST can be computed for any process model in linear time and it is unique [15, 12].

A node in an RPST corresponds to a fragment of one out of four types: *trivial*, *polygon*, *bond* or *rigid*. A *trivial* consists of a single edge. A *polygon* represents a sequence of fragments. A *bond* corresponds to a subgraph where all child fragments are adjacent to the entry and exit nodes of the fragment. Any other case is a *rigid* fragment. We use the prefixes T, P, B and R to designate the type of fragment. For example fragment B1 is a bond. This bond appears in three different places (its occurrences are thus exact clones). Meanwhile, bonds B2 and B4 could be considered as approximate clones, depending on the user-defined distance threshold. Similarly, one level above, R1, R2 and R3 could also be considered as approximate clones.

Figures 1(a)–(c) present sample process fragments extracted from models in the SAP Reference Model [6]. For sake of clarity, only SESE fragments with at least four vertices are identified in the figures, surrounded by a dashed rectangle. Moreover, Figure 1(d) shows a simplified (tree) representation of the RPST of each fragment in Figures 1(a)–(c). Consider the process fragment shown in Figure 1(a). We can observe that this fragment contains three bonds, viz. B1, B2 and B3; two non-trivial polygons, viz. P1 and P2; and a rigid fragment, viz. R1. Furthermore, the rigid R1 is the root fragment, having B1, P1, and P2 as children. Finally, polygon P1 is parent of bonds B2 and B3.

The process models and fragments in this paper use EPC as the underlying notation. However, the presented techniques are notation-independent.

2.2 RPSDAG

The RPSDAG [14] is an index structure designed for an efficient and accurate identification of exact clones in a collection of process models. Conceptually, it can be thought

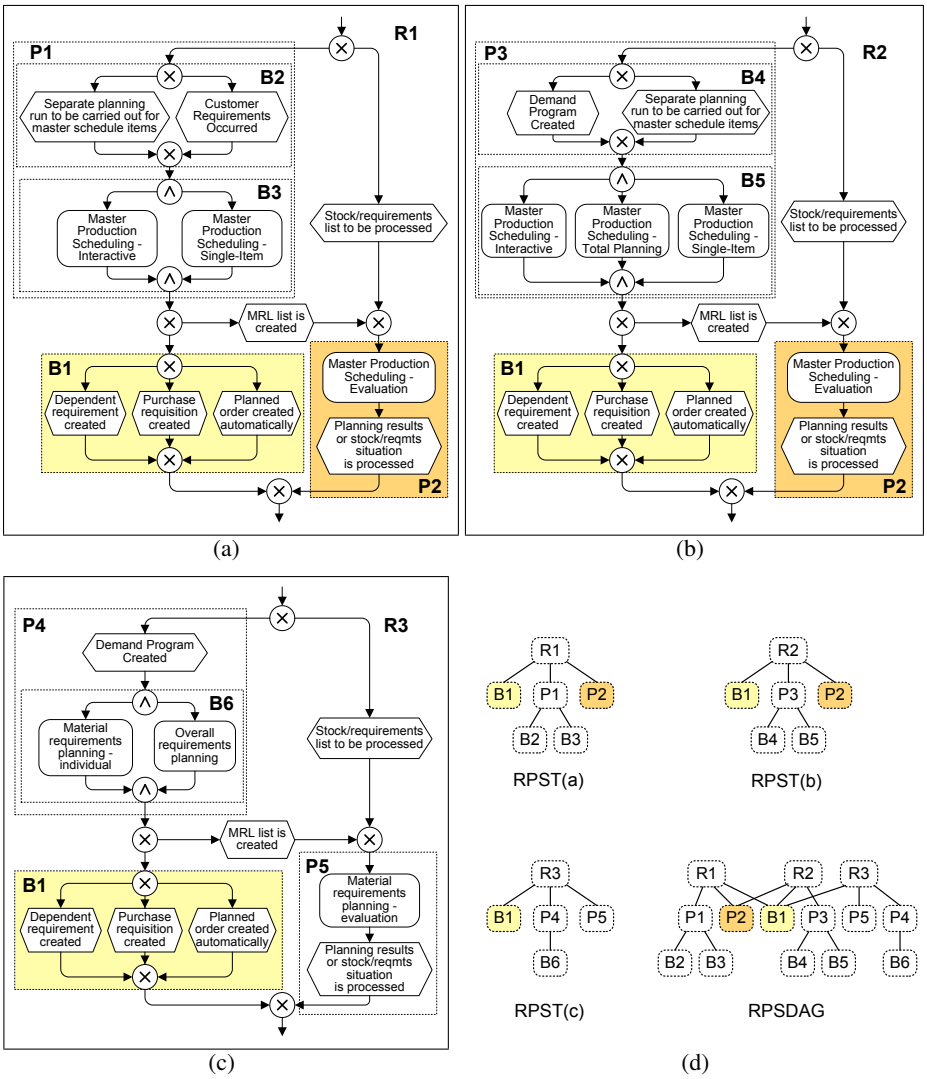

Fig. 1. Process fragments extracted from the SAP Reference Model.

as the union of a set RPSTs represented. A node in the RPSDAG corresponds to a SESE fragment of a model in the collection, whereas edges encode the containment relation among SESE fragments. Importantly, each fragment only appears once in the RPSDAG. Thus, if a fragment appears multiple times, in the same RPST or in different RPSTs, it is factored out and represented only once in the RPSDAG. For example, Figure 1(d) shows the RPSTs and the RPSDAG of the process fragments presented in Figures 1(a)–(c). Note that fragments B1 and P2 are represented only once in the RPSDAG. A node in the RPSDAG that has more than one parent is an exact clone fragment.

The RPSDAG is built incrementally. When a new process model is added to the collection, the corresponding RPST is computed and merged into the existing RPSDAG. The RPSDAG implementation described in [14] incorporates several optimizations that make it scalable to real-life repositories of process models with hundreds of models. In addition to identifying exact clones, the RPSDAG allows us to determine if a process fragment is contained in another – a feature we will use during clustering.

2.3 Process Model Similarity

The similarity of process models specified in a graph-based notation can be measured on the basis of three complementary aspects: the labels attached to tasks, events and other model elements; their graph structure; or their execution semantics. In this paper, we adopt a measure that combines structural and label similarity and that has been shown to be correlated with perceived similarity [3]. This measure is defined over an abstract representation of process models based on labelled graphs, as follows.

Definition 3 (Process graph). *Let $\mathcal{L}$ be a set of labels. A* (business) process graph *H is a tuple (V, E, λ) where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $\lambda : V \to \mathcal{L}$ is a function that maps vertices to labels.*

The adopted similarity measure is based on the well-known graph-edit distance [10]. The graph edit distance of two graphs is the minimal set of edit operations required to transform one graph into the other. There are three edit operations: vertex substitution, vertex insertion/deletion and edge insertion/deletion. A vertex substitution refers to the fact that a vertex in one of the graphs is mapped to a vertex in the other graph. To define a valid vertex substitution, we require a notion of vertex similarity. In this respect, we consider that vertices are matched according to their label similarity measured in terms of string-edit distance, denoted as $Sim_{led}(label_1, label_2)$.[2] A vertex substitution is only allowed if the similarity between their labels is above a user-defined threshold (e.g. 0.6). Whenever a vertex in a graph is not matched to any vertex in the other graph, it is considered as either inserted in one graph or deleted in the other one. Similarly, an edge insertion (or deletion) operation is required for each edge that cannot be mapped to an edge in the other graph. This intuition is formalized as follows.

Definition 4 (Normalized process graph edit distance [2]). *Let $H_1 = (V_1, E_1, \lambda_1)$ and $H_2 = (V_2, E_2, \lambda_2)$ be two process graphs. Let $M : V_1 \nrightarrow V_2$ be a partial injective mapping that maps vertices of H_1 to vertices of H_2. Moreover, let $subv$ be the set of substituted vertices, i.e., $\forall v \in subv : v \in dom(M) \cup cod(M)$, $skipv$ the set of skipped vertices, i.e., $\forall v \in skipv : v \notin dom(M) \cup cod(M)$, and $skipe$ the set of skipped edges, i.e., $\forall (v, w) \in skipe : v \notin dom(M) \cup cod(M) \vee w \notin dom(M) \cup cod(M)$. The* normalized graph edit distance *induced by the mapping M is:*

$$Dist^{M}_{GED}(H_1, H_2) = \frac{wskipv \cdot fskipv + wskipe \cdot fskipe + wsubv \cdot fsubn}{wskipv + wskipe + wsubv}$$

[2] Other measures of label similarity (e.g. semantic ones) can be used as discussed in [2].

where $wskipv$, $wskipe$ and $wsubv$ are relative weights in the range $[0..1]$ assigned to each graph-edit operation, $fskipv$ is the fraction of skipped vertices, $fskipe$ the fraction of skipped edges, and $fsubv$ the average distance between substituted vertices, defined as $fskipv = \frac{|skipv|}{|V_1|+|V_2|}$, $fskipe = \frac{|skipe|}{|E_1|+|E_2|}m$ and $fsubv = \frac{2 \cdot \sum_{(v,w) \in M} 1 - Sim_{led}(\lambda_1(v), \lambda_2(w))}{|E_1|+|E_2|}$, where $Dist_{led}$ is the string-edit distance between vertex labels.

Finally, the normalized graph-edit distance between H_1 and H_2, written $Dist_{GED}(H_1, H_2)$, is the smallest $Dist^M_{GED}(H_1, H2)$ across all mappings M.

A $Dist_{GED}$ of 0 means that the process graphs are identical, while a $Dist_{GED}$ of 1 implies that the process graphs are completely dissimilar.

The problem of computing the graph-edit distance is NP-Complete [10]. In this paper, we adopt a fast greedy heuristic described in [2]. Still, despite the fact that we use a greedy heuristic, the computation of the $Dist_{GED}$ is expensive. Accordingly, before computing the actual $Dist_{GED}$ between two graphs, we first calculate a lower-bound of it. When this lower-bound is above threshold τ (cf. Definition 1), we do not need to compute $Dist_{GED}$ to determine if two fragments are approximate clones. In this way, we avoid unnecessary calculations when clustering. The lower-bound is obtained from the following observations. First, we take the largest of the two graphs (i.e. the one with more nodes and more edges). Say that H_1 is larger than H_2 (otherwise we revert the roles). Now, assuming that H_1 is a subgraph of H_2, all vertices of H_1 can be substituted by vertices of H_2, all edges of H_1 are matched with edges of H_2, and no vertices are substituted. The only differences come from the vertices and edges of H_2 that are not in H_1. Thus, $fskipv = \left| \frac{|V_1|-|V_2|}{|V_1|+|V_2|} \right|$, $fskipe = \left| \frac{|E_1|-|E_2|}{|E_1|+|E_2|} \right|$ and $fsubv = 0$. These are lower-bound values. If the assumption that H_1 is not a subgraph of H_2 is violated, then the graph-edit distance will necessarily be greater because it entails additional differences. Thus, we conclude that $Dist_{GED}(H_1, H_2)$ is greater than the one obtained by feeding the above lower-bound values of $fskipv$, $fskipe$ and $fsubv$ into the equation for $Dist^M_{GED}(H_1, H2)$ in Definition 4. Note that if the graphs have equal size, the obtained lower-bound is zero – which is not useful.

3 Approximate Clones Clustering

In order to operationalize the two approaches proposed in the introduction, we reviewed various clustering algorithms and selected two of them which allowed us, with minor adaptations, to fulfill our requirements. These are the *Density-Based Spatial Clustering of Applications with Noise (DBSCAN)* [13] for the first approach, and the *Hierarchical Agglomerate Clustering (HAC)* [13] for the second approach. In both algorithms, described below, we assume that the distance between every possible pair of fragments has been pre-computed and stored in a *distance matrix*. This matrix only stores the distance $Dist_{GED}$ of Definition 4 for a pair of fragments if this is within the user-defined threshold τ, and if the two fragments do not contain one another (*non-containment relationship*). For all other fragment pairs, it stores ∞.

3.1 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

In the first approach we propose to standardize a set of clones towards a medoid fragment. Given a cluster, a medoid is an element of the cluster that is the closest to the center of the cluster. The medoid does not necessarily coincide with the center of the cluster (called *centroid*) since in our problem the distance between the medoid and all other cluster elements is not the same, but is bounded by the user-defined threshold. A well-known algorithm that is built upon this principle is DBSCAN. DBSCAN creates clusters based on the density of *neighborhoods*. Given a set of objects O, the neighborhood of an object $o \in O$ is the set of fragments $N_o = \{o_i \in O \mid d(o, o_i) \leq \epsilon\}$, where $d(o, o_i)$ is a distance measure between o and o_i and ϵ is the neighborhood radius. A *core object* is an object whose $|N_o| \geq Size_{min}$, where $Size_{min}$ is the minimum cluster size (we observe that a core object is contained in its neighborhood since its distance with itself is 0). Thus, we have to specify two parameters for this algorithm: neighborhood radius and minimum cluster size. In our case, the neighborhood radius coincides with the used-defined distance threshold τ, whereas we can fix $Size_{min}$ to 2 to retrieve clusters of at least two fragments. Moreover, we use the notion of graph-edit distance $Dist_{GED}$ as the distance measure between two objects.

Standard DBSCAN identifies all core objects of a given dataset and considers their neighborhoods as initial clusters. If two core objects are within each other's neighborhood, their neighborhoods are merged into a single cluster. On the other hand, if an object does not belong to the neighborhood of any core object, it is marked as *noise*. Our adaptation of DBSCAN is described in Algorithm 1. Given the set of process fragments G extracted from the RPSDAG, the algorithm repeats the clustering process (Steps 2–14) until all fragments in G have been checked whether they are core objects. At the beginning of each iteration, a random fragment f is removed from G and marked as "processed". The neighborhood N_f of f is computed (Step 3), and if f is a core object the fragments in N_f are removed from G and from $Noise$ (Step 5), and added to a new cluster C (Step 6). Otherwise f is treated as noise and another fragment is extracted from G. The algorithm then *expands* cluster C by checking whether there are core objects in C whose neighborhoods can be merged with C. This is done by iterating over all fragments in N_f except f, via a set M_C. For a fragment m in M_C that has not been processed, its neighborhood N_m is computed (Step 8) to determine whether m is itself a core object. If so, before merging its neighborhood with C, we check whether there is still a medoid s whose distance with all other fragments of the combined cluster is within τ (Step 10), otherwise we will create clusters whose fragments are far apart from each other to be standardized. In case of merging, the fragments in N_m are removed from G and added, except m, to M_C (Step 11), so that they can be checked whether they are core objects. If N_m cannot be merged with C, m is added back to G so that it can be eventually processed again (Step 12). In fact, N_m may form a cluster by itself or be merged with some other cluster.

A fragment's neighborhood is constructed using the distance matrix. Given the non-containment relation enforced by this matrix, a fragment cannot be in the neighborhood of a core object that contains or is contained by it. Still, it is possible to include two related fragments in a neighborhood if they are both sufficiently similar to the core object. To prevent this, we retrieve the set of all the ascendants and descendants of a fragment

Algorithm 1: DBSCAN Clustering

Input: Set G of process fragments.
Output: The sets of clusters ($Clusters$) and noise ($Noise$).

1 Initialize $Clusters$ and $Noise$ to empty sets.
2 Remove a fragment f from G and mark f as "processed".
3 Retrieve the neighborhood N_f.
4 If $|N_f| < Size_{min}$, add f to $Noise$, then go to 2.
5 Remove N_f from G and from $Noise$.
6 Initialize a new cluster C in $Clusters$ with N_f, and a new set M_C to $N_f \setminus \{f\}$.
7 Remove a fragment m from M_C.
8 If m is not "processed", mark m as "processed" and retrieve N_m.
9 If $N_m \geq Size_{min}$
10 If there is a fragment $s \in C \cup N_m$ such that for all $p \in C \cup N_m$ $Dist_{GED}(s, p) \leq \tau$
11 Remove N_m from G and $Noise$ and add N_m to C and $N_m \setminus \{m\}$ to M_C.
12 Else, mark m as "unprocessed" and add it to G.
13 If $M_C \neq \varnothing$ go to 7.
14 If $G \neq \varnothing$ go to 2.

by computing its transitive closure on the RPSDAG, and add to the neighborhood the fragment in the transitive closure that is the nearest to the core object (the original fragment may thus be discarded in favor of one of its ascendants or descendants). Further, we mark all other fragments in the transitive closure as "visited" for that cluster, so that these fragments will not be included in any neighborhood of that cluster.

The complexity of Algorithm 1 is dominated by that of the neighborhood computation (Steps 3 and 8), and by that of the merging condition (Step 10). Neighborhood computation for a fragment f requires at most $|G| - 1$ lookups in the distance matrix. The exploration of the transitive closure of each neighbor of f requires further $|G| - 1$ lookups (retrieving the transitive closure of an RPSDAG node is linear on the RPSDAG size, which is bounded by $|G|$). Similarly, the merging condition requires $|G| - 1$ lookups in the distance matrix for all members of a cluster. As the main loop is repeated $|G|$ times, the overall complexity of Algorithm 1 is $O(|G|^3)$. This is higher than the complexity of standard DBSCAN, which is $O(|G|^2)$ [13]. That said, in our experience the algorithm showed to be efficient (cf. Section 4). In fact, the search space is greatly reduced by the cutoff conditions used when computing the distance of clusters, i.e. the distance threshold τ and the non-containment relationship. The result is that the distance matrix is highly sparse, but the sparsity depends on intrinsic characteristics of the process model collection. Further, we store each computed neighborhood so that it can be reused when reprocessing a core object whose neighborhood has not been merged.

3.2 Hierarchical Agglomerate Clustering (HAC)

In the second approach, a set of clones can be standardized by selecting any fragment in the group as a reference and standardizing all other fragments towards this reference fragment. In other words, we require that every pair of fragments in a cluster has a distance below the threshold τ. This goal can be straightforwardly mapped to the strategy

followed by the basic hierarchical agglomerative clustering method [13]. This clustering method starts with singleton clusters and iteratively combine the pair of clusters that is found to be the closest among all other possible pairs. The process of merging continues until there is only one cluster left.

One key issue is the definition of the distance between two clusters, which needs to be recomputed after every cluster merging. Several possibilities are available: taking the smallest distance between fragments in one of the clusters to the fragments in the other one, known as *single link*; taking the farthest distance, referred to as *complete link*; among others. It can be easily see that the complete link strategy suites well to our purposes, as it allows to identify the cluster mergings that will not meet the requirement of keeping a distance below the threshold τ. Note that the identification of such situation can be accomplished ahead of time. The intuition is captured in the following definition.

Definition 5 (Distance of clusters under complete link strategy). *Let C_i and C_j be clusters in the dendrogram built by a hierarchical clustering algorithm, and τ be the similarity threshold among fragments of C_i and fragments of C_j. Moreover, let $\mathcal{F}(C)$ be a function that returns the set of fragments associated to C, inductively defined as follows: (BASE) if C is a leaf node in the dendrogram, C is a singleton and refers to a single fragment, say f, then $\mathcal{F}(C) = \{f\}$; (STEP) if C is an intermediate node then $\mathcal{F}(C) = \cup_{c \in C} \mathcal{F}(c)$. The distance of clusters C_i and C_j, denoted as $Dist(C_i, C_j)$, can defined as follows.*

$$\begin{cases} \infty & \text{if } \exists f \in \mathcal{F}(C_i), g \in \mathcal{F}(C_j) : g \subseteq f \vee f \subseteq g \\ \infty & \text{if } max_{f \in \mathcal{F}(C_i), g \in \mathcal{F}(C_j)} \, Dist_{GED}(f, g) > \tau \\ max_{f \in \mathcal{F}(C_i), g \in \mathcal{F}(C_j)} \, Dist_{GED}(f, g) & \text{otherwise} \end{cases}$$

We note that the distance of two clusters is set to ∞ when there exist one fragment in the first cluster which is in containment relationship with another fragment in the second cluster. Moreover, when farthest distance between fragments of both clusters is above the threshold τ, the distance is set to ∞. In the two previous cases, we are meeting the constraints described in Definitions 1 and 2. Finally, the farthest distance between fragments of both clusters is reported as the distance of the clusters, only when the value is less or equal to the threshold τ. Algorithm 2 corresponds to the modified version of the basic hierarchical agglomerative method adapted for clustering approximate clones.

Algorithm 2 can be divided into two parts. Step 1 and 2, initialize the set of singleton clusters, stores them in $TopClusters$ and initializes the distance matrix between clusters (according to Definition 5). The remaining steps correspond to the main loop. In Step 3, a pair of clusters is selected such that their distance is found to be the smallest among all other possible pairs. If the distance of such pair is ∞ or there is only one cluster left then the algorithm stops. In Step 4, a new cluster is created to hold the union the previously selected pair. In Step 5, the distance matrix is updated (according to Definition 5), by removing the pair clusters previously selected and adding the newly created cluster.

The algorithm starts with a working set of $|G|$ clusters. In every iteration, two clusters are removed and a new one is added. Hence, the size of the working set decreases monotonically. The algorithm stops when $|TopClusters| = 1$ or before if the entire distance matrix $\mathcal{D}$ is filled with ∞.

Algorithm 2: Hierarchical Agglomerative Clustering

Input: Set G of process fragments.
Output: The set of maximal clusters, viz. $TopClusters$.

1 For each $f \in G$ create a singleton cluster. Initialize $TopClusters$ to contain all singleton clusters.
2 Using the distance matrix between fragments, calculate the initial distance matrix between clusters in $TopClusters$, i.e. $\mathcal{D}[i,j] \leftarrow Dist(C_i, Cj)$, where $C_i, C_j \in TopClusters$.
3 In the distance matrix $\mathcal{D}$, select a pair of clusters $C_i, C_j \in TopClusters$ such that their distance is the minimum. Stop if no such pair exists, i.e. either all distances in $\mathcal{D}$ are ∞ or $|TopClusters| = 1$.
4 Combine clusters C_i and C_j to form a new cluster C_{ij}. Remove clusters C_i and C_j from $TopClusters$. Add cluster C_{ij} to $TopClusters$.
5 Update matrix $\mathcal{D}$ by adding the distance between cluster C_{ij} and all other clusters in $TopClusters$.
6 Go to 3.

The complexity of Algorithm 2 is dominated by the maintenance of the distance matrix (i.e., Steps 2 and 5), which has an initial size of $O(|G|^2)$. As the main loop is repeated $O(|G| - 1)$ times, the worst-case upper bound of the complexity is of $O(|G|^3)$ [13]. The same simplifications of the search space that we used for DBSCAN apply to HAC (distance cutoff and non-containment). Also this algorithm has shown to be efficient in our experience.

4 Evaluation

We implemented the two algorithms as a Java console application and evaluated them in a twofold manner. First, we performed a descriptive analysis of approximate clone clusters in two industrial process model collections in order to assess the potential usefulness of the techniques. Secondly, we conducted an experiment to measure the accuracy of the technique at retrieving fragments resulting from copy/pasting and subsequent independent changes. Both experiments make use of a measure of cluster quality intended to capture the potential benefits of standardization.

4.1 Cluster quality measure

The proposed techniques are aimed at retrieving clusters of fragments that can be standardized into a common fragment. Such a standardization activity entails a certain effort and brings in certain benefits – in the form of less duplication and thus smaller total repository size. We contend that clusters that have a higher benefit-to-cost ratio are most likely to be candidates for standardization. In particular, if a cluster of approximate clones has emerged from copy/pasting of a fragment followed by independent changes of the copied fragments, it is likely to have a high benefit-to-cost ratio, provided that the changes made are not considerable.

To operationalize the benefit-to-cost ratio as a measure of cluster quality, we need to define a cost measure and a benefit measure. The cost of standardizing the fragments of a cluster into a single fragment is determined by many factors, some of them exogenous to the process models themselves. However, we contend that this cost is proportional to the amount of elementary changes that will be made to the fragments in order to standardize them to one common subprocess. Indeed, each elementary change will require a certain amount of effort to ensure that the execution of the process is adapted to this change. Accordingly, we hereby use the absolute GED ($Dist_{AGED}(H_1, H_2)$) defined in the same way as $Dist_{GED}(H_1, H_2)$ in Definition 4 but replacing $fskipv$ and $fskipe$ with $|skipv|$, $|skipe|$ respectively, and removing the denominator in the definition of $fsubv$. In other words, we count actual number of edit operations as opposed to fraction of edit operations relative to total size. We do not used the normalized GED in this context ($Dist_{GED}$), because this normalized version is not reflective of the number of operations required to standardize the fragments. Instead, $Dist_{GED}$ is reflective of the percentage difference shared between two models.

In the case of clusters produced using DBSCAN, there is a designated medoid that serves as a reference. Thus, the cost of standardizing the cluster is the sum of the distances between each fragment in the cluster and the medoid (m), i.e. $\sum_{f \in C} Dist_{AGED}(f, m)$. In the case of clusters produced using hierarchical clustering, every fragment in the cluster could potentially be used as the "medoid" towards which all fragments would be standardized. Assuming that the aim is to maximize the benefit-to-cost ratio, we will pick as medoid the fragment that will yield the highest benefit-to-cost ratio (see below).

The benefit of standardizing and refactoring a cluster into a subprocess is proportional to the amount of reduction in duplication, which in turn reflects itself in a reduction in size of the overall repository. This size reduction is equal to the sum of the sizes of the fragments in the cluster (since they are removed) to which we subtract the size of the medoid – since this medoid becomes a new subprocess – and the number of fragments – since each cluster is replaced by a "call activity" to the subprocess. In other words, the benefit of standardizing a cluster is $\sum_{f \in C} |f| - |m| - |C|$.

Given the above, we define the benefit-to-cost ratio of a cluster obtained with the DBSCAN method as $BCR(C) = \frac{\sum_{f \in C} Dist_{AGED}(f,m)}{\sum_{f \in C} |f|-|m|-|C|}$. In the case of hierarchical clustering, we define the benefit-to-cost ratio of a cluster as the maximum of BCR(C) across all fragments in the cluster.

4.2 Potential usefulness assessment

We assessed the potential usefulness of the approximate clone clustering techniques using two datasets. The first dataset is the SAP R/3 reference model [6]. It contains 595 models with sizes ranging from 5 to 119 nodes (average 22.28). The second dataset is taken from an insurance company under condition of anonymity. It contains 363 models ranging from 4 to 461 nodes (average 27.12). We first computed the RPSDAG for both datasets and post-processed them by factoring out all exact clones using the technique presented in [14]. This yielded 2348 non-trivial fragments for the SAP dataset (11.47 average size) and 2037 non-trivial fragments for the insurance dataset (16.58 av-

erage size). We then applied the two clustering methods independently – having eliminated exact clones to avoid double-counting. The clustering algorithms were run with a $Dist_{GED}$ threshold of 0.4.

All tests were run on a PC with a dual core Intel processor, 1.8GHz, 4GB memory, running Microsoft Windows 7 and Oracle Java Virtual Machine v1.6. The cluster computation is dominated by the computation of the distance matrix which took 26.3 mins for the SAP dataset and 2.69 hours for the insurance dataset. The time for clustering itself is negligible in comparison. The longer time taken for the insurance dataset is justified by the size of its fragments – much larger than those in the SAP dataset (e.g. the largest fragment in the insurance dataset is a rigid with 461 nodes whereas the largest SAP fragment contains 117 nodes).

Figure 2 plots the histograms of distribution of cluster sizes for the two datasets. For the SAP dataset we retrieved a total of 364 clusters with DBSCAN (with sizes ranging from 2 to 5 clusters) and 335 clusters for HAC (sizes between 2 and 13), while for the insurance dataset we retrieved 243 clusters with DBSCAN (sizes between 2 and 6) and 309 clusters with HAC (sizes between 2 and 10). This confirms the intuition that real-life process model repositories contain a large number of approximate clone clusters, and thus that copy/pasting of fragments across process models is a very common practice. Looking at the size distribution, for both datasets the majority of the clusters retrieved by the two algorithms contain between 2 and 8 fragments, with the largest clusters having 2 fragments. This suggests that copy/pasting is typically limited to 6-8 copies per fragment.

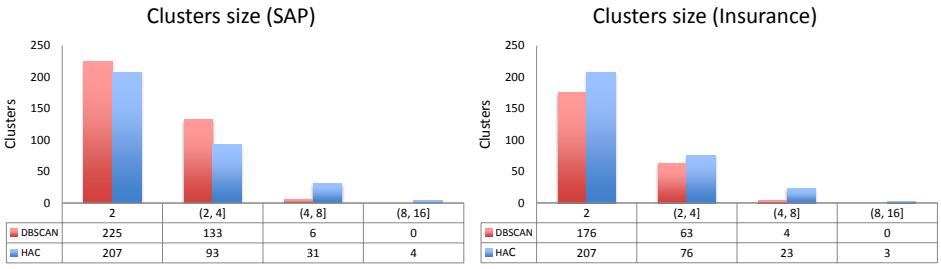

Fig. 2. Number of clusters vs clusters size for both algorithms.

Figure 3 shows the histograms of distributions of BCR for both datasets. We observe that in general none of the techniques performs better than the other, since for the SAP dataset we achieve higher BCRs for HAC than for DBSCAN, whilst for the insurance dataset it is the other way around. This suggests that depending on the type of the repository, one of the two techniques might be more appropriate than the other.

4.3 Retrieving copy/pasted fragments

The second experiment aimed to evaluate the accuracy of the clustering techniques with respect to the task of retrieving clusters of clones that have emanated from a single original fragment by means of copy/pasting followed by independent changes to the duplicated fragments. We did so by simulating a situation where new fragments are inserted in an existing process model repository by copying a master fragment across

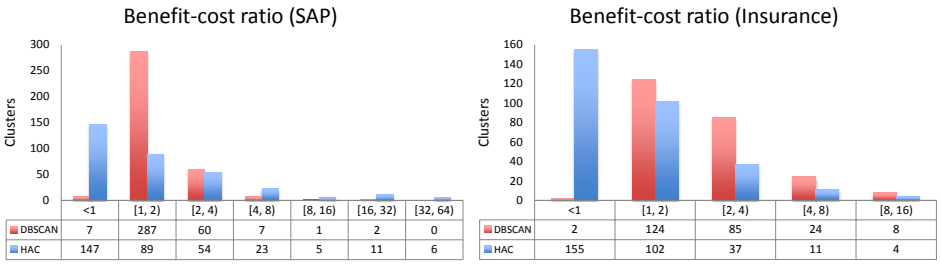

Fig. 3. Number of clusters vs benefit/cost ratio for both algorithms.

various models of the repository, after doing minor changes. We randomly extracted 50 fragments from the two datasets used in the previous experiment, such that they were sufficiently different from each other (pairwise graph-edit distance above 70%).

To test the accuracy of the DBSCAN algorithm, we used these 50 fragments as "seeds" to generate 50 artificial clusters by producing from 2 to 10 variants for each seed, and grouping each seed with its variants in a cluster. We obtained a total of 311 fragments in 50 clusters. Seed variants were obtained by applying simple change operations (edge/node removal or insertion), such that the graph-edit distance between a variant and its seed was no more than 40% – the same threshold that we used in the first experiment. The clusters' size ranged from 3 to 10 fragments (average 6.35). We then generated 300 process models from the two existing datasets, such that none of these models contained any of the seed fragments, and we randomly inserted the 311 fragments into these models such that a model would contain from 0 to 2 fragments. We then extracted the RPSDAG from this dataset and clustered the retrieved fragments using our DBSCAN. The algorithm retrieved 328 clusters. We matched each artificial cluster with the retrieved fragment that yielded the maximum *FScore* [17]. FScore is the harmonic mean of the recall and precision of a retrieved cluster with respect to (w.r.t.) an artificial cluster. Precisely, given an artificial cluster l and a retrieved cluster s, the FScore of s w.r.t. l is $F(s,l) = \frac{2 \cdot R(s,l) \cdot P(s,l)}{R(s,l)+P(s,l)}$ where $R(s,l)$ and $P(s,l)$ are the recall and precision of s w.r.t. l.

In order to measure the overall quality of the algorithm, we then computed the *weighted average FScore* (F_{wa}) [17]. F_{wa} is the maximum FScore of each artificial cluster weighted against the combined size of all artificial clusters. Let L be the set of artificial clusters and S the set of retrieved clusters. Then $F_{wa} = \sum_{l=1}^{L} \frac{|l|}{|L|} F(l)$, where $F(l) = \max_{s \in S} F(s,l)$.

We repeated the same experiment for the HAC algorithm. In order to ensure that all fragments in an artificial cluster have pairwise graph-edit distance within the 40% threshold, we used a *random walk* approach. From each seed we generated a variant with graph-edit distance of at most 0.4. We chose one of these two fragments and generated another variant such that its distance to both fragments was at most 0.4, and so on until we generated from 2

	Recall				**Precision**				F_{wa}
	min	max	avg	std	min	max	avg	std	
DBSCAN	0.17	1	0.71	0.37	0.2	1	0.89	0.24	0.73
HAC	0.1	1	0.82	0.25	0.17	1	0.84	0.33	0.77

Table 1. Various quality metrics for the two algorithms.

to 10 variants for each cluster. This led to a total of 289 fragments in 50 clusters, with sizes ranging from 3 to 10 fragments (average 5.8). We inserted these fragments in the collection of 300 process models that we generated in the previous step, and then clustered the fragments retrieved from the RPSDAG of this collection using HAC. This led to 295 clusters.

The results for both algorithms are reported in Table 1. Besides F_{wa}, this table reports the minimum, maximum, average and std. deviation of recall and precision for the best-matched retrieved cluster for each artificial cluster. The accuracy of the two algorithms is partly affected by the presence of approximate clones that exist in the generated process model collections, besides those that have been generated artificially. Despite this, the results show high F_{wa} (0.73 for DBSCAN and 0.77 for HAC), as well as high average precision and recall for both algorithms, demonstrating the accuracy of the algorithms. None of the algorithms clearly outperforms the other.

Finally, we used the above data to evaluate the ranking accuracy of the BCR. For each algorithm, we plotted a ROC curve by ordering the retrieved clusters from the highest to the lowest BCR. In these curves, we considered a retrieved cluster as a true positive if it had a recall of 1, and as a true negative otherwise. The curves, shown in Fig. 4, show that the clusters with highest BCR are indeed those that most closely match the synthetically generated clusters. This result is confirmed by the Area Under the Curve which is 0.72 for DBSCAN and 0.89 for HAC (both with asymptotic significance less than 0.05).

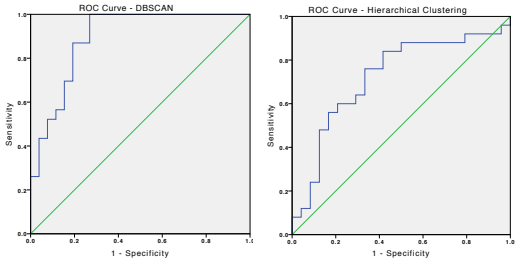

Fig. 4. ROC curves for both algorithms.

5 Related Work

Clone detection in software repositories has been an active field of research for several years [7]. However in this field focus has been on exact software clone detection.

In the field of model-driven engineering, approximate clone detection has been investigated in [1] and [11]. In [1] the authors present *CloneDetective*, a method for detecting clones in large repositories of Simulink/TargetLink models from the automotive industry. Models are partitioned into connected components and compared pairwise using a heuristic subgraph matching algorithm. According to [11], CloneDetective suffers from low inaccuracy and low degree of completeness in detection, mainly due to the fact that small clones are absorbed by larger clone pairs. In other words, the algorithm tends to find as large clones as possible, whereas in our approach we allow related fragments to belong to different clusters, in order to allow users to choose the abstraction level at which to standardize. Moreover, this method is not very sensitive to approximate clones having small differences. These cases commonly result from copy/pasting and as such they should not be discarded. Moreover, they yield low standardization costs making them easy to standardize. The work in [11] overcomes these problems by

proposing two methods for exact and approximate matching of clones. In particular, the second method, namely *aScan*, represents graphs by a set of vectors built from graph features: e.g. path lengths and vertex in/out degrees. An empirical study shows that this feature-based approximate matching improves pre-processing and running times, while keeping a high precision. Despite these advantages, the method proposed in [11] does not fulfill our requirements: The resulting clones may be non-SESE fragments and the identified clusters do not necessarily satisfy any of the properties in Definition 2.

Refactoring process model collections has been investigated in [14, 4, 16]. In [14], we described a technique to find fragments that are equal across different process models, so that they can be factored out in separate subprocess. In this paper, we assume that all such exact clones have already been factored out, but we reuse the RPSDAG structure that we built in [14] to identify hierarchical dependencies among fragments in different process models. In [4], process fragments that are sufficiently similar to each other are identified. In contrast to our work, fragment similarity is exclusively based on label similarity rather than a combination of label and structural similarity. Also, fragments are considered pairwise and no clustering takes place. This approach can help analysts detect overlap between process models, however no support is offered to standardize these similar fragments such that they can be refactored. In [16], eleven process model refactoring techniques are identified and evaluated. Extracting process fragments as subprocesses is one of the techniques identified. Our work addresses the problem of identifying opportunities for such "fragment extraction" and provides an actual implementation and experimentation. In addition, [16] does not consider clustering.

Clustering of process models has been dealt with in [5] and [9]. In both cases process models are clustered rather than process fragments leading to a small number of clusters. Using fragments instead of process models is more complex, but for the purposes of standardization and reuse it is more suitable as a fragment may be shared between process models, while the rest of these models may be quite different.

In [8] an approach is described to synthesize the most representative process model out of a collection of variants. This work is complementary to ours in that it could be used after clustering has been applied in order to synthesize the centroid of a cluster. However, this is not the approach we followed as this may likely lead to an artificially created centroid which does not represent an actual fragment occurring in a process model. The presence of such an artificial fragment could cause problems for a business analyst when trying to standardize a cluster.

6 Conclusion

This paper presented two techniques for retrieving clusters of approximate clones for possible refactoring into shared subprocesses. Experiments showed that both clustering techniques, coupled with the proposed measure of cluster quality (benefit-to-cost ratio), are able to accurately retrieve clusters resulting from copy/pasting activity followed by independent modifications to the copied fragments. Hence, the proposed techniques could be used to re-consolidate copies of a fragment. A descriptive analysis of clones in two industrial process model repositories put into evidence a proliferation of approximate clones of varying sizes and benefit-to-cost ratio.

The evaluation is limited in at least two respects. First, clustering and cluster ranking accuracy are evaluated based on synthetic data – albeit generated via perturbations of real-world fragments. The retrieved clusters may not be reflective of the types of clusters that analysts would find most suitable for standardization and refactoring. Addressing this limitation requires a realistic "golden standard", for example, one resulting from a manual assessment of cluster quality by domain experts. This is a direction for future work. A second limitation of the study is that only two repositories were used to evaluate the potential benefit of the proposed techniques. In one case one technique led to higher overall benefit-to-cost ratio, while the reverse was observed in the second case. Further evaluation is needed to determine in what cases one technique should be preferred over the other. Finally, the evaluation could be extended to include other clustering techniques.

References

1. F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *ICSE*, 2008.
2. R.M. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. In *BPM*, volume 5701 of *LNCS*. Springer, 2009.
3. R.M. Dijkman, M. Dumas, B.F. van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Inf. Syst.*, 36(2):498–516, 2011.
4. R.M. Dijkman, B. Gfeller, J.M. Küster, and H. Völzer. Identifying refactoring opportunities in process model repositories. *Information & Software Technology*, 53(9):937–948, 2011.
5. J.-Y. Jung and J. Bae. Workflow clustering method based on process similarity. In *ICCSA*, volume 3981 of *LNCS*. Springer, 2006.
6. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.
7. R. Koschke. Identifying and Removing Software Clones. In *Software Evolution*. Springer, 2008.
8. C. Li, M. Reichert, and A. Wombacher. The minadept clustering approach for discovering reference process models out of process variants. *IJCIS*, 19(3-4):159–203, 2010.
9. J. Melcher and D. Seese. Visualization and clustering of business process collections based on process metric values. In *SYNASC*. IEEE, 2008.
10. B.T. Messmer. *Efficient Graph Matching Algorithms*. PhD thesis, Switzerland, 1995.
11. N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen. Complete and Accurate Clone Detection in Graph-based Models. In *ICSE*, pages 276–286. IEEE, 2009.
12. A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified Computation and Generalization of the Refined Process Structure Tree. In *WSFM*, 2010.
13. P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
14. R. Uba, M. Dumas, L. García-Bañuelos, and M. La Rosa. Clone detection in repositories of business process models. In *BPM*, pages 248–264, 2011.
15. J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.
16. Barbara Weber, Manfred Reichert, Jan Mendling, and Hajo A. Reijers. Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486, 2011.
17. Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *CIKM*, pages 515–524. ACM, 2002.