Fast Detection of Exact Clones in Business Process Model Repositories

Marlon Dumas[a], Luciano García-Bañuelos[a], Marcello La Rosa[b,c], Reina Uba[a]

[a] *University of Tartu, Estonia*
[b] *Queensland University of Technology, Brisbane, Australia*
[c] *NICTA Queensland Lab, Brisbane, Australia*

Abstract

As organizations reach higher levels of business process management maturity, they often find themselves maintaining very large process model repositories, representing valuable knowledge about their operations. A common practice within these repositories is to create new process models, or extend existing ones, by copying and merging fragments from other models. We contend that if these duplicate fragments, a.k.a. *exact clones*, can be identified and factored out as shared subprocesses, the repository's maintainability can be greatly improved. With this purpose in mind, we propose an indexing structure to support fast detection of clones in process model repositories. Moreover, we show how this index can be used to efficiently query a process model repository for fragments. This index, called RPSDAG, is based on a novel combination of a method for process model decomposition (namely the Refined Process Structure Tree), with established graph canonization and string matching techniques. We evaluated the RPSDAG with large process model repositories from industrial practice. The experiments show that a significant number of non-trivial clones can be efficiently found in such repositories, and that fragment queries can be handled efficiently.

Keywords: Process model repository, clone detection, indexing structure, query

1. Introduction

Organizations engaged in long-term business process management programs need to deal with repositories of hundreds or even thousands of process models, with sizes ranging from dozens to hundreds of elements per model [1, 2]. For example, the SAP reference model contains over 600 business process models, while Suncorp, a large Australian insurer, maintains a repository of over 3,000 process models [3]. Tool vendors nowadays distribute reference model repositories (e.g. the IT Infrastructure Library – ITIL) with over a thousand process models each. Such models are used to document and to communicate internal procedures, to guide the development of IT systems, or to support business improvement projects, among other uses.

Email addresses: marlon.dumas@ut.ee (Marlon Dumas), luciano.garcia@ut.ee (Luciano García-Bañuelos), m.larosa@qut.edu.au (Marcello La Rosa), reinak@ut.ee (Reina Uba)

While highly valuable, such collections of process models raise a maintenance challenge [4]. This challenge is amplified by the fact that process models are generally maintained and used by stakeholders with varying skills, responsibilities and goals, sometimes distributed across independent organizational units. One problem that arises as repositories grow is that of managing overlap across models. In particular, process model repositories tend to accumulate duplicate fragments over time, as new process models are created by copying and merging fragments from other models. Experiments conducted during this study have put into evidence a large number of clones in three process model repositories from industrial practice. This situation is akin to that observed in source code repositories, where significant amounts of duplicate code fragments – known as *code clones* – are accumulated over time [5].

Cloned fragments in process models raise several issues. Firstly, clones make individual process models larger than they need to be, thus affecting their comprehensibility. Secondly, clones are modified independently, sometimes by different stakeholders, leading to unwanted inconsistencies across models. Finally, process model clones hide potential efficiency gains. Indeed, by factoring out cloned fragments into separate subprocesses, and exposing these subprocesses as shared services, companies may reap the benefits of larger resource pools.

Detecting clones by comparing process models in a pairwise manner – using subgraph isomorphism algorithms – would be inefficient in the context of repositories with hundreds of process models. Accordingly, indexes are needed to speed up the clone discovery process.

In this setting, this article addresses the problem of retrieving all clones in a process model repository that can be refactored into shared subprocesses. Specifically, the contribution of the article is an index structure, namely the *RPSDAG*, that provides operations for inserting and deleting models, as well as an operation for retrieving all clones in a repository that meet the following requirements:

- All retrieved clones must be single-entry, single-exit (SESE) fragments, since subprocesses are invoked according to a call-and-return semantics.
- All retrieved clones must be *exact* clones so that every occurrence can be replaced by an invocation to a single (shared) subprocess. While identifying *approximate* clones could be useful in some scenarios, approximate clones cannot be directly refactored into shared subprocesses, and thus fall outside the scope of this study.
- All retrieved clones are *maximal*, in the sense that each of them has multiple non-identical enclosing SESE regions. This "maximality" criterion is desirable because once we have identified a clone, every SESE fragment strictly contained in this clone is also a clone, but we do not wish to return all such non-maximal sub-clones.
- Retrieved clones must have at least two nodes (no "trivial" clones).

Identifying clones in a process model repository boils down to identifying fragments of a process model that are isomorphic to other fragments in the same or in another model. Hence, we need a method for decomposing a process model into fragments and a method for testing isomorphism between these fragments. Accordingly, the RPSDAG is built on top of two pillars: (i) a method for decomposing a process model into SESE fragments, namely the *Refined Process Structure Tree (RPST)* decom-

position; and (ii) a method for calculating canonical codes for labeled graphs. These canonical codes reduce the problem of testing for graph isomorphism between a pair of graphs, to a string equality check. These two techniques however need some adaptations in order to fit the requirements of the RPSDAG. Firstly the RPST does not retrieve all possible SESE regions in a model. Some SESE regions are hidden inside others, like for example sub-sequences hidden inside larger sequences. Secondly, naive methods for calculating canonical codes for labeled graphs have problems scaling up, especially when there are multiple nodes with duplicate labels (or unlabeled nodes). Process models contain "gateways" and gateways are generally not labeled. In this paper, we propose several optimizations that take advantage of the specific structure of process models in order to scale up the calculation of canonical codes.

As a byproduct, we show how the RPSDAG can also be used to efficiently answer "fragment queries", that is, queries aimed at retrieving all occurrences of a given model fragment in a repository.

The rest of the paper is structured as follows. Section 2 introduces the concepts of RPST and canonical code and discusses how they are used to address the problem at hand. Next, Section 3 describes the RPSDAG, including its insertion and deletion algorithms. This section also shows how the RPSDAG can be used to efficiently retrieve all occurrences of a given process model fragment in a repository. Section 4 presents an empirical evaluation of the RPSDAG. Finally, Section 5 discusses related work while Section 6 draws conclusions.

2. Background

This section introduces the two basic ingredients of the proposed technique: the Refined Process Structure Tree (RPST) and the code-based graph indexing.

2.1. RPST

The RPST [6] is a parsing technique that takes as input a process model and computes a tree representing a hierarchy of SESE fragments. Each fragment corresponds to the subgraph induced by a set of edges. A SESE fragment in the tree contains all fragments at the lower level, but fragments at the same level are disjoint. As the partition is made in terms of edges, a single vertex may be shared by several fragments.

Each SESE fragment in an RPST is of one of four types [7]. A *trivial* (T) fragment consists of a single edge. A *polygon* (P) fragment is a sequence of fragments. A *bond* corresponds to a fragment where all child fragments share a common pair of vertices. Any other fragment is a *rigid*.

Figure 1(a) presents a sample process model for which the RPST decomposition is shown in the form of dashed boxes. A simplified view of the RPST is shown in Fig. 2.

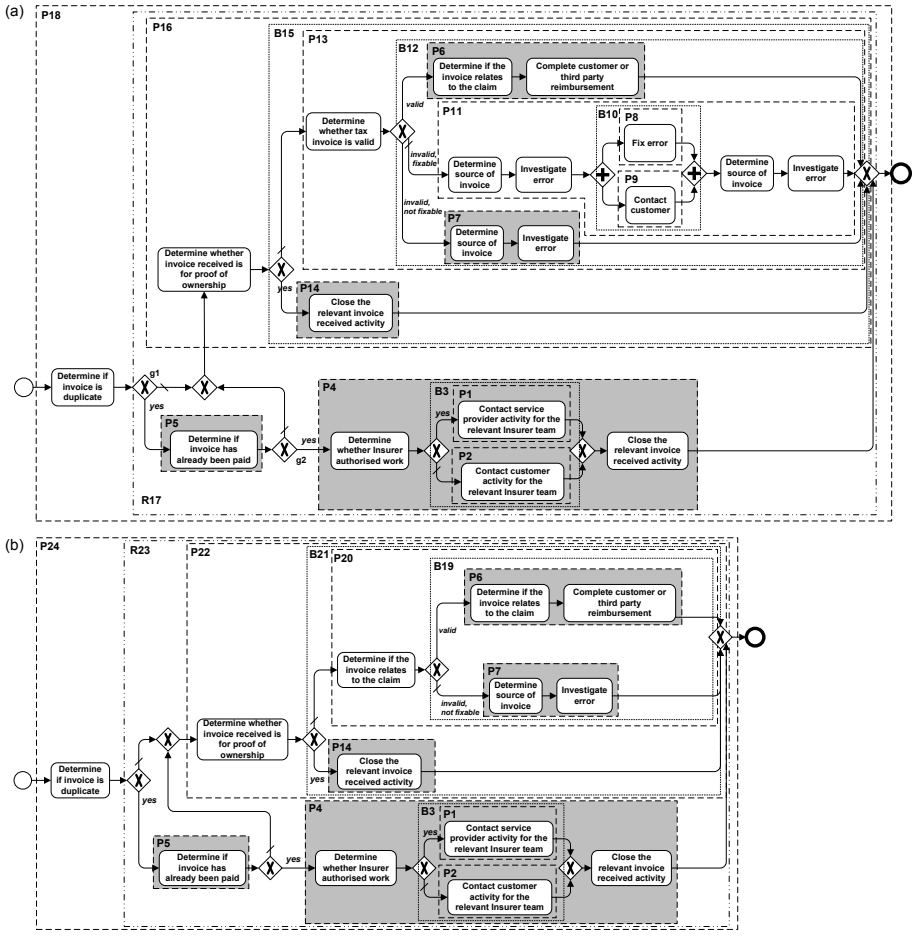

Figure 1: Excerpt of two process models of an insurance company.

We observe that the root fragment corresponds to polygon P18, which in turn contains rigid R17 and a set of trivial fragments (simple edges) that are not shown in order to simplify the figure. R17 is composed of polygons P16, P5, P4, and so forth.

In this setting, every fragment in the RPST is a potential candidate when identifying clones. Consider the two models presented in Figure 1, where polygons P1, P2, P4, P5, P6, P7 and P14, and bond B3 have been highlighted to ease their identification. Out of these we note that polygon P4 has a size greater than one node and is a maximal clone (i.e., it is not embedded in a larger clone). This is an example of clone that we wish to detect. B3 on the other hand is not a maxi-

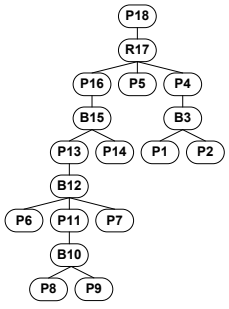

Figure 2: RPST of the process model in Fig. 1(a).

mal clone as it is always embedded in P4, whereas P5 is a trivial clone. Thus, B3 and P4 are examples of clones we are not interested in detecting. We also observe that the sequence of tasks ["Determine source of invoice", "Investigate error"] appears within polygon P11. This same sequence is identical to polygon P7. Thus, we can say that this sequence is a *sub-polygon* shared by P11, B12 and B19. Moreover, we note that this sequence occurs twice under polygon P11: once before and once after B10. In this case we will say that there are two *sibling* occurrences of sub-polygon P7 under polygon P11. Similarly, B19 is contained in B12, thus B12 is an example of *sub-bond*. It should be noted that the RPST per se does not explicitly represent sub-polygons and sub-bonds because the RPST is designed to capture only maximal polygons and maximal bonds. Thus, sub-polygons and sub-bonds need to be extracted separately.

Although the models used as examples are encoded in BPMN, the proposed technique can be applied to other graph-based modeling notations. To achieve this notation-independence, we adopt the following graph-based representation of process models:

Definition 2.1 (Process Graph). A *Process Graph* is a labelled connected graph $G = (V, E, l)$, where:

- V is the set of vertices.
- $E \subseteq V \times V$ is the set of directed edges (e.g. representing control-flow relations).
- $l : V \rightarrow \Sigma^*$ is a labeling function that maps each vertex to a string over alphabet Σ. We distinguish the following special labels: $l(v) =$ "start" and $l(v) =$ "end" are reserved for start events and end events respectively; $l(v) =$ "xor-split" is used for vertices representing xor-split (exclusive decision) gateways; similarly $l(v) =$ "xor-join", $l(v) =$ "and-split" and $l(v) =$ "and-join" represent merge gateways, parallel split gateways and parallel join gateways respectively. For a task node t, $l(t)$ is the label of the task.

This definition can be extended to capture other types of BPMN elements by introducing additional types of nodes (e.g. a type of node for inclusive gateways, data objects etc.). Organizational information such as lanes and pools can be captured by attaching dedicated attributes to each node (e.g. each node could have an attribute indicating the pool and lane to which it belongs) [8]. In this paper, we do not consider sub-processes, since each sub-process can be indexed separately for the purpose of clone identification.

2.2. Canonical labeling of graphs

Our approach for graph indexing is an adaptation of the approach proposed in [9]. The adaptations we make relate to two specificities of process models that differentiate them from the class of graphs considered in [9]: (i) process models are directed graphs; (ii) process models can be decomposed into an RPST.

Following the method in [9], our starting point is a matrix representation of a process graph encoding the vertex adjacency and the vertex labels, as defined below.

Definition 2.2 (Augmented Adjacency Matrix of a Process Graph). Let $G = (V, E, l)$ be a Process Graph, and $\boldsymbol{v} = (v_1, \ldots, v_{|V|})$ a total order over the elements of V. The adjacency matrix of G, denoted as $\boldsymbol{A}$, is a $(0, 1)$-matrix such that $A_{i,j} = 1$ if and only if $(v_i, v_j) \in E$, where $i, j \in \{1 \ldots |V|\}$. Moreover, let us consider a function $h : \Sigma^* \rightarrow \mathbb{N} \setminus \{0, 1\}$ that maps each vertex label to a unique natural number greater than 1. The *Augmented Adjacency Matrix* $\boldsymbol{M}$ of G is defined as: $\boldsymbol{M} = \text{diag}(\ h(l(v_1)),\ \ldots,\ h(l(v_{|V|}))\ ) + \boldsymbol{A}$

Given the augmented adjacency matrix of a process graph (or a SESE fragment therein), we can compute a string (hereby called a *code*) by concatenating the contents of each cell in the matrix from left to right and from top to bottom. For illustration, consider graph G in Figure 3(a), which is an abstract view of fragment B3 of the running example (cf. Figure 1(a)). For convenience, next to each vertex we show the unique vertex identifier (e.g. v_1), the corresponding label (e.g. $l(v_1) =$"A"), and the numeric value associated with the label (e.g. $h(l(v_1)) = 2$). Assuming the order $\boldsymbol{v} = (v_1, v_2, v_3, v_4)$ over the set of vertices, the matrix shown in Figure 3(b) is the adjacency matrix of G. Figure 3(c) is the diagonal matrix built from $h(l(\boldsymbol{v}))$ whereas Figure 3(d) shows the augmented adjacency matrix $\boldsymbol{M}$ for graph G. It is now clear why 0 and 1 are not part of the codomain of function h, i.e. to avoid clashes with the representation of vertex adjacency. Figure 3(e) shows a possible permutation of $\boldsymbol{M}$ when considering the alternative order $\boldsymbol{v'} = (v_1, v_4, v_2, v_3)$ over the set of vertices.

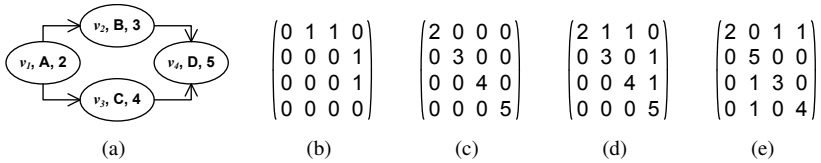

Figure 3: (a) a sample graph, (b) its adjacency matrix, (c) its diagonal matrix with the vertex label codes, (d) its augmented adjacency matrix, and (e) a permutation of the augmented matrix

Next, we transform the augmented adjacency matrix into a code by scanning from left-to-right and top-to-bottom. For instance, the matrix in Figure 3(d) can be represented as "2.1.1.0.0.3.0.1.0.0.4.1.0.0.0.5". This code however does not uniquely represent the graph. If we chose a different ordering of the vertices, we would obtain a different code. To obtain a unique code (called *canonical code*), we need to pick the code that lexicographically "precedes" all other codes that can be constructed from an augmented adjacency matrix representation of the graph. Conceptually, this means that we have to consider all possible permutations of the set of vertices and compute a code for each permutation, as captured in the following definition.

Definition 2.3 (Graph Canonical Code). Let G be a process graph, $\boldsymbol{M}$ the augmented adjacency matrix of G. The *Graph Canonical Code* is the smallest lexicographical string representation of any possible permutation of matrix $\boldsymbol{M}$, that is:

$$\text{code}(\boldsymbol{M}) = \text{str}(\boldsymbol{P^T M P}) \big| \boldsymbol{P} \in \boldsymbol{\Pi} \wedge \ \forall \boldsymbol{Q} \in \boldsymbol{\Pi}, \boldsymbol{P} \neq \boldsymbol{Q} : \text{str}(\boldsymbol{P^T M P}) \prec \text{str}(\boldsymbol{Q^T M Q})$$

where:

- $\boldsymbol{\Pi}$ *is the set of all possible permutations of the identity matrix* $\boldsymbol{I}_{|M|}$
- *str*$(\boldsymbol{N})$ *is a function that maps a matrix* $\boldsymbol{N}$ *into a string representation.*

Consider the matrices in Figures 3(d) and 3(e). The code of the matrix in Figure 3(e) is "2.0.1.1.0.5.0.0.0.1.3.0.0.1.0.4". This code is lexicographically smaller than the code of the matrix in Figure 3(d) ("2.1.1.0.0.3.0.1.0.0.4.1.0.0.0.5"). If we explored all vertex permutations and constructed the corresponding matrices, we would find that "2.0.1.1.0.5.0.0.0.1.3.0.0.1.0.4" is the canonical code of the graph in Figure 3(a).

Enumerating all vertex permutations is unscalable (factorial on the number of vertices). Fortunately, optimizations can be applied by leveraging the characteristics of the graphs at hand. Firstly, by exploiting the nature of the fragments in an RPST, we can apply the following optimizations:

- The code of a polygon is computed in linear time by concatenating the codes of its contained vertices in the (total) order implied by the control-flow.
- The code of a bond is computed in linear time by taking the entry gateway as the first vertex, the exit gateway as the last vertex, and all vertices in-between in lexicographic order based on their labels. Since these vertices in-between do not have any control-flow dependencies among them, duplicate labels do not pose a problem.

In the case of a rigid, we start by partitioning its vertices into two subsets: vertices with distinct labels, and vertices with duplicate labels. Vertices with distinct labels are deterministically ordered in lexicographic order. Hence we do not need to explore any permutations between these vertices. Instead, we can focus on deterministically ordering vertices with duplicate labels. Duplicate labels in process models arise in two cases: (i) there are multiple tasks (or events) with the same label; (ii) there are multiple gateways of the same type (e.g. multiple "xor-splits") that cannot be distinguished from one another since gateways generally do not have labels. To distinguish between multiple gateways, we pre-process each gateway g by computing the tasks that immediately precede it and the tasks that immediately follow it within the same rigid fragment, and in doing so, we skip other gateways found between gateway g and each of its preceding (or succeeding) tasks. We then concatenate the labels of the preceding tasks (in lexicographic order) and the labels of the succeeding tasks (again in lexicographic order) to derive a new label s_g. The s_g labels derived in this way are used to order multiple gateways of the same type within the same rigid. Consider for example g_1 and g_2 in Figure 1a and let s1 = "Determine if invoice has already been paid", s2="Determine whether invoice received is for proof of ownership" and s3="Determine whether Insurer authorized work". We have that s_{g_1} = "s1.s2" while s_{g_2} = "s1.s3.s2". Since s3 precedes s2, gateway g_2 will always be placed before g_1 when constructing an augmented adjacency matrix for R17. In other words, we do not need to explore any permutation where g_1 comes before g_2. Even if task "Determine if invoice is duplicate" precedes g_1 this is not used to compute s_{g_1} because this task is outside rigid R17. To ensure unicity, vertices located outside a rigid should not be used to compute its canonical code.

A similar approach is used to order tasks with duplicate labels within a rigid. This "label propagation" approach allows us to considerably reduce the number of permutations we need to explore. Indeed, we only need to explore permutations between multiple vertices if they have identical labels and they are preceded and followed by vertices that also have the same labels. The worst-case complexity for computing the canonical code is still factorial, but on the size of the largest group of vertices inside a rigid that have identical labels and identical predecessors and successors' labels.

3. Clone Detection Method

This section introduces the RPSDAG, including its underlying data structure, insertion and deletion algorithms and its support for "all-clone queries" and "fragment queries".

3.1. Index structure and clone detection

The RPSDAG is a directed acyclic graph representing the union of the RPSTs of a collection of process models. The key feature of the RPSDAG is that each SESE fragment is represented only once, even if it appears multiple times in the RPST of a given process model or across multiple RPSTs. When a new process model is inserted into an RPSDAG, its RPST is computed and traversed bottom-up. For each fragment, it is checked if this fragment already exists in the RPSDAG. If it does, the fragment is not inserted, but instead, the existing fragment is reused. For instance, consider an RPSDAG constructed from the process model shown in Figure 1a. Imagine that the process model in Figure 1b is inserted into this RPSDAG, meaning that the fragments in its RPST are inserted into the RPSDAG one by one from bottom to top. When the RPST node corresponding to polygon P4 is reached, we detect that this fragment already exists. Rather than inserting the fragment, an edge is created from its parent fragment (R23) to the existing node P4.

The RPSDAG is designed to be implemented on top of standard relational databases. Specifically, all the data required by the RPSDAG is stored in three tables: Codes(Code, Id, Size, Type), Roots(GraphId, RootId) and RPSDAG(ParentId, ChildId, Weight).

Table Codes contains the canonical code for each RPST fragment of an indexed graph. Column "Id" assigns a unique identifier to each indexed fragment, column Code gives the canonical code of a fragment, column Size is the number of vertices in the fragment, and column Type denotes the type of fragment ("p" for polygon, "r" for rigid and "b" for bond, plus the special types "sp" for sub-polygon and "sb" for sub-bond which are discussed later). The "Id" is auto-generated by incrementing a counter every time that a new code is inserted into the Codes table. Strictly speaking, the "Id" is redundant given that the code uniquely identifies each fragment. However, this "Id" gives us a shorter way of uniquely identifying a fragment. For optimization reasons, when the canonical code of a new RPST node is constructed, we use the short identifiers of its child fragments as labels in the diagonal of the augmented adjacency matrix, instead of using the child fragment's canonical codes, which are much longer. For example fragment P13 in Figure 1a. contains two child fragments: node "Determine whether tax invoice is valid" and bond *B12*. Instead of using the label "Determine whether tax invoice is valid" and the canonical code of B12 to compute the canonical code of P13, we attach a short numerical identifier to label "Determine whether tax invoice is valid" and we reuse the short identifier of B12 when constructing the canonical code of P13.

Table Roots contains the Id of each indexed graph and the Id of the root fragment for that graph. Table RPSDAG is the main component of the index and is a combined representation of the RPSTs of all the graphs in the repository. Since multiple RPSTs may share fragments (these are precisely the clones we look for), the RPSDAG table represents a Directed Acyclic Graph (DAG) rather than a tree. Each tuple of this table consists of a parent fragment Id, a child fragment Id and a Weight representing the number of occurrences of the fragment identified by the child Id within the fragment identified by the parent Id. Figure 4 shows an extract of the tables representing the two graphs in Figure 1. Here,s the fragment Ids have been derived from the fragment names in the Figure (e.g. the Id for P4 is 4) and the codes have been hidden for the sake of readability.

Code	Id	Size	Type
-	1	1	p
-	2	1	p
-	3	4	b
-	4	6	p
-	5	1	p
-	6	2	p
...	...	...	...
-	17	26	r
-	18	27	p
...	...	...	...
-	24	21	p

ParentId	ChildId	Weight
3	1	1
3	2	1
4	3	1
...	...	...
11	7	2
12	6	1
12	11	1
12	7	1
...	...	...
17	4	1
17	5	1
...	...	...
18	17	1
19	6	1
19	7	1
...	...	...
23	4	1
23	5	1
...	...	...

GraphId	RootId
1	18
2	24

Figure 4: RPSDAG schema for the process models in Fig. 1.

Looking at the RPSDAG, we can immediately observe that the maximal clones are those child fragments whose sum of weights is greater than one. This means that they occur more than once, either in different parents or within the same parent. For example, P4 occurs within R17 and R23. Accordingly, we can retrieve all clones in an RPSDAG by means of the following SQL query.

```
SELECT RPSDAG.ChildId, Codes.Size, SUM(RPSDAG.Weight)
  FROM RPSDAG, Codes
WHERE RPSDAG.ChildId = Codes.Id AND Codes.Size >= 2
GROUP BY RPSDAG.ChildId, Codes.Size
HAVING SUM(RPSDAG.Weight) >= 2;
```

This query retrieves the Id, size and number of occurrences of fragments that have at least two parent fragments. We observe that if a fragment appears multiple times in the indexed graphs, but always under the same parent fragment, it is not a maximal clone. For example, fragment P2 in Figure 1 always appears under B3, and B3 always appears under P4. Thus, neither P2 nor B3 are maximal clones, and the above query does not retrieve them. On the other hand, the query identifies P4 as a maximal clone since it has two parents (cf. tuples (17,4,1) and (23,4,1) in table RPSDAG in Fig. 4).

Also, as per the requirements spelled out in Section 1, the query returns only fragments with at least 2 nodes. This is because we are interested in identifying clones that can be refactored into separate subprocesses and is not worth replacing single nodes with subprocesses. For example, even if there are two tuples with child Id 5 in the example RPSDAG of Fig. 4, the query will not return clone P5 as it has a single node.

3.2. Insertion

Algorithm 1 describes the procedure for inserting a new process graph into an indexed repository. Given an input graph, the algorithm first computes its RPST with function ComputeRPST() which returns the RPST's root node. Next, procedure InsertFragment is invoked on the root node to update tables Codes and RPSDAG. This returns the id of the root fragment. Finally, a tuple is added to table Roots with the id of the inserted graph and that of its root node.

Algorithm 1: Insert Graph

procedure InsertGraph(Graph m)
  RPST $root \Leftarrow$ ComputeRPST(m)
  $rid \Leftarrow$ InsertFragment($root$)
  $Roots \Leftarrow Roots \cup \{(\text{NewId}(), rid)\}$ // NewId() generates a fresh id

Algorithm 2: Insert Fragment

procedure InsertFragment(RPST f) returns RPSDAGNodeId
  {RPST, RPSDAGNodeId} $C \Leftarrow \varnothing$
  foreach RPST *child in* GetChildren(f) **do**
    $C \Leftarrow C \cup \{(child, \text{InsertFragment}(child))\}$
  $code \Leftarrow$ ComputeCode(f, C)
  $(id, type) =$ InsertNode($code, f$)
  foreach (cf, cid) *in* C **do**
    $weight =$ GetWeight(id, cid)
    $RPSDAG \Leftarrow RPSDAG \setminus \{(id, cid, weight)\} \cup \{(id, cid, weight+1)\}$
  if $type =$*"p"* **then** InsertSubPolygons($id, code, f$)
  else if $type =$*"b"* **then** InsertSubBonds($id, code, f$)
  return id

Procedure InsertFragment (Algorithm 2) inserts an RPST fragment in table RPSDAG. This algorithm performs a depth-first search traversal of the RPST. Nodes are visited in postorder, i.e. a node is visited after all its children have been visited. When a node is visited, its canonical code is computed – function ComputeCode – based on the topology of the RPST fragment and the codes of its children (except of course for leaf nodes whose labels are their canonical codes). Next, procedure InsertNode is invoked to insert the node in table Codes, returning its id and type. This procedure (Algorithm 3) first checks if the node already exists in Codes via function GetIdSizeType(). If it exists, GetIdSizeType() returns the id, size and type associated with that code, otherwise it returns the tuple (0,0,""). In this latter case, a fresh id is created for the node at hand, then its size and type are computed via functions ComputeSize and ComputeType, and a new tuple is added in table Codes. Function ComputeSize returns the sum of the sizes of all child nodes or 1 if the current node is a leaf. Once the node id and type have been retrieved, procedure InsertFragment adds a new tuple in

table RPSDAG for each children of the visited node if that tuple did not already exist, otherwise it increases its weight.

Algorithm 3: Insert Node

```
procedure InsertNode(String code, RPST f) returns
(RPSDAGNodeId, String)
    (id, size, type) ⇐ GetIdSizeType(code)
    if (id, size, type) = (0, 0, "") then
        id ⇐ NewId()
        size ⇐ ComputeSize(code)
        type ⇐ ComputeType(f)
        Codes ⇐ Codes ∪ {(code, id, size, type)}
    return (id,type)
```

Algorithm 4: Insert SubPolygons

```
procedure InsertSubPolygons(RPSDAGNodeId id, String code, RPST f)
    foreach (zcode, zid, zsize, ztype) in Codes such that ztype ="p" ∨ "sp"
    and zid ≠ id do
        {String} LCS ⇐ ComputeLCS(code, zcode)
        foreach lcode in LCS do
            (lid, ltype) = InsertSubNode(lcode, f)
            if lid ≠ id then RPSDAG ⇐ RPSDAG ∪ {(id, lid, 1)}
            if lid ≠ zid then RPSDAG ⇐ RPSDAG ∪ {(zid, lid, 1)}
            foreach sid in GetChildrenIds(lid) do
                RPSDAG ⇐
                RPSDAG \ {(id, sid, 1), (zid, sid, 1)} ∪ {(lid, sid, 1)}
            InsertSubPolygons(lid, lcode)
```

If the visited node is a polygon, procedure InsertSubPolygons is invoked at the end of InsertFragment. Procedure InsertSubPolygons is used to identify common sub-polygons and to factor them out as separate nodes in the RPSDAG. Let us consider again the example in Figure 1. As mentioned in Section 2, the sequence of activities ["Determine source of invoice", "Investigate error"] – which is identical to polygon P7 – occurs inside P11: once before and once after B10. These occurrences should be recognized as occurrences of polygon P7. Hence, P7 is a *sub-polygon* shared by polygons P11, B12 and B19. Procedure InsertSubPolygons identifies such sub-polygons and materializes them as explicit nodes in the RPSDAG. This explains the presence of tuple (11,7,2) in the RPSDAG of Figure 4.

To see how sub-polygons are created, let us consider the two polygons P_1 and P_2 in Fig. 5, where $\text{code}(P_1) = B_2.a.B_1.w.z.a.B_1.c$ and $\text{code}(P_2) = a.B_1.c.d.a.B_1.w.z$. These two polygons share bond B_1 as common child, while bond B_2 is a child of P_1 only. However, at a closer look, their canonical codes share three Longest Com-

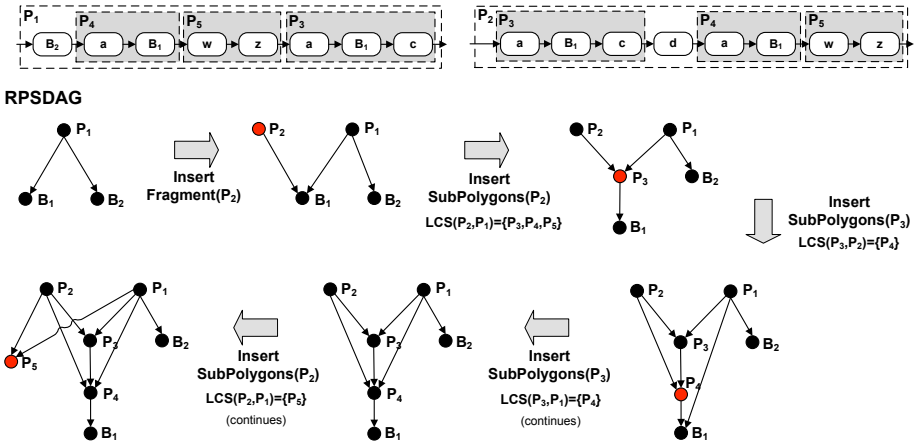

Figure 5: Two sub-polygons and the corresponding RPSDAG.

mon Substrings (LCS), namely $a.B_1.c$, $a.B_1$ and $w.z$.[1] These common substrings represent common *sub-polygons*, and thus clones that may be refactored as separate subprocesses.

Assume P_1 is already stored in the RPSDAG with children B_1 and B_2 (first graph in Fig. 5) and we now want to store P_2. We invoke procedure InsertFragment(P_2) and add a new node in the RPSDAG, with B_1 as a child (second graph in Fig. 5). Since P_2 is a polygon, we also need to invoke InsertSubPolygons(P_2). This procedure (Algorithm 4) retrieves all polygons from Codes that are different than P_2 (in our case there is only P_1). Then, for each such polygon, it computes all the non-trivial LCSs between its code and the code of the polygon being inserted. This is performed by function ComputeLCS() that returns an ordered list of LCSs starting from the longest one. In the example at hand, the longest substring is $a.B_1.c$. This substring can be seen as the canonical code of a "shared" sub-polygon between P_2 and P_1. To capture the fact that this shared sub-polygon is a clone, we insert a new node P_3 in the RPSDAG with canonical code $a.B_1.c$ (unless such a node already exists, in which case we reuse the existing node) and we insert an edge from P_2 to P_3 and from P_1 to P_3. We also add a new tuple for P_3 in Codes and set its type to "sp" to keep track of sub-polygons (needed when removing a model from the repository – see Algorithm 5). If the node already exists, we change its type to "sp" in Codes. This is done by function InsertSubNode() which is similar to InsertNode() with the only difference that it sets the type of the new node to "sp", while if the node exists it changes its type to "sp".

In order to avoid redundancy in the RPSDAG, the new node P_3 then "adopts" all the common children that it shares with P_2 and with P_1. These nodes are retrieved by function GetChildrenIds(), which simply returns the ids of all child nodes of P_3: these

[1] An LCS of strings S_1 and S_2 is a substring of both S_1 and S_2, which is not contained in any other substring shared by S_1 and S_2. LCSs of size one are not considered for obvious reasons. LCSs can efficiently be identified using a *suffix tree* [10], see e.g. [11] for a survey of LCS algorithms

will either be children of P_1 or P_2 or both.[2] In our example, there is only one common child, namely B_1, which is adopted by P_3 (see third graph in Fig. 5).

It is possible that the newly inserted node also shares sub-polygons with other nodes in the RPSDAG. So, before continuing to handle the remaining LCSs between P_1 and P_2, we invoke procedure InsertSubPolygons recursively over P_3. In our example, the code of P_3 shares the substring $a.B_1$ with both P_2 and P_1 (after excluding the code of P_3 from the codes of P_2 and P_1 since P_3 is already a child of these two nodes). Thus we create a new node P_4 with code $a.B_1$ as a child of P_3 and P_2, and we make P_4 adopt the common child B_1 (see fourth graph in Fig. 5). We repeat the same operation between P_3 and P_1 but since this time P_4 already exists in the RPSDAG, we simply remove the edge between P_1 and B_1, and add an edge between P_1 and P_4 (fifth graph in Fig. 5). Then, we resume the execution of InsertSubPolygons(P_2) and move to the second LCS between P_1 and P_2, i.e. $a.B_1$. Since this substring has already been inserted into the RPSDAG as node P_4, nothing is done. This process of searching for LCSs is repeated until no more non-trivial common substrings can be identified. In the example at hand, we also add sub-polygon $w.z$ between P_1 and P_2 (last graph in Fig. 5). At this point, we have identified and factored out all maximal sub-polygons shared by P_1 and P_2, and we can repeat the above process for other polygons in the RPSDAG whose canonical code shares a common substring with that of P_1.

Sometimes there may be multiple overlapping LCSs of the same size. For example, given the codes of two polygons $a.b.c.d.e.f.a.b.c$ and $a.b.c.k.b.c.d$, if we extract one substring (say $a.b.c$), we can no longer extract the second one ($b.c.d$). In these cases we locally choose one of the overlapping LCS based on the number of occurrences of an LCS within the two strings in question. If they have the same number of occurrences, we randomly choose one. In the example above, $a.b.c$ has the same size of $b.c.d$ but it occurs 3 times, so we pick $a.b.c$ and extract the corresponding sub-polygon.

Procedure InsertSubPolygons also handles the cases where the polygon to be inserted is a sub-polygon or super-polygon of an existing polygon. These are just special cases of shared sub-polygon detection.

Coming back to Algorithm 2, if the node to be added is a bond, procedure InsertSubBonds is called in order to identify common *sub-bonds* and factor them out as separate nodes. Take for example bond B19 in Fig. 1b. This bond is actually a sub-bond of B12 in Fig. 1a, since B12 contains B19 plus P11. Thus we want to add a parent-child relation from B12 to B19 in the RPSDAG, and change the type of B19 in Codes to "sb" (sub-bond).

Procedure InsertSubBonds is similar to InsertSubPolygons, except that instead of detecting sub-polygons using ComputeLCS, we detect sub-bonds. Detecting sub-bonds is simpler than detecting sub-polygons. Since the order of the children within a bond is irrelevant, we simply need to compute the intersection between the children of the bond being inserted and the set of children of each indexed bond. If the size of this intersection is greater than one, it means that the bond being inserted shares a sub-bond with an already indexed bond. Naturally, we only compare the bond being inserted with an indexed bond if they have the same types of split and join behavior.

[2]They are children of one of the two polygons only, if the other polygon is a sub-polygon of the former.

For example, it does not make sense to find a sub-bond shared between a bond that has XOR-splits at its ends, and a bond that has AND-splits at its end.

If we find that the intersection between the bond being inserted B and an already indexed bond B' contains at least two nodes[3] we have effectively found a sub-bond SB consisting of the children shared between B and B'. If sub-bond SB overlaps with any sub-bond already indexed under B', then we do not insert it because we are not interested in retrieving overlapping clones – if we did, one of the two overlapping clones could not be extracted as a shared sub-process because the other clone would already contain some of its elements. If on the other hand there is no overlap between SB and any of the existing sub-bonds under B', we insert SB under B'. The mechanism to insert sub-bond SB is the same as for sub-polygons, i.e. a new node is created in the RPSDAG capturing the shared sub-bond. Given that procedure InsertSubBonds is very similar to InsertSubPolygons, we do not show its pseudo-code.

Sometimes, a fragment may be contained multiple times in the same parent fragment, like for example sub-polygon P7 which is contained twice within P11 in Figure 1. These *sibling clones* are captured by attribute "Weight" to each edge of the RPSDAG, representing the number of times a child node occurs inside a given parent node. A Weight greater than 1 indicates a case of sibling clones. Our implementation of the RPSDAG does not store the attribute Weight for every edge in the RPSDAG but only for edges that have a weight of at least two. This optimization is achieved by separating the RPSDAG table into two tables: one containing edges with a weight of one and another for edges with a weight greater than one. The rationale is that only a small fraction of edges have a weight greater than one. This is purely an implementation-level optimization. At the conceptual level, we view the RPSDAG table as a single table.

3.3. Deletion

Algorithm 5 shows the procedure for deleting a graph from an indexed repository. This procedure relies on another procedure for deleting a fragment, namely DeleteFragment (Algorithm 6). The DeleteFragment procedure performs a depth-first search traversal of the RPSDAG s visiting the nodes in post-order. Nodes with at most one parent are deleted, because they correspond to fragments that appear only in the deleted graph. Deleting a node entails deleting the corresponding ok in table Codes and deleting all tuples in the RPSDAG table where the deleted node corresponds to the parent Id. If a fragment has two or more parents, the traversal stops along that branch since this node and its descendants must remain in the RPSDAG. After invoking procedure DeleteFragment, the graph itself is deleted from table Roots through its root Id.

Before completing the DeleteGraph procedure, procedure CleanRPSDAG is triggered (see Algorithm 7). This procedure cleans up the RPSDAG by removing those sub-fragments (i.e. sub-polygons and sub-bonds) which have been added with procedures InsertSubPolygons and InsertSubBonds but that are now left with a single parent as part of executing DeleteGraph. There is no reason to keep these sub-fragments in the RPSDAG since they may prevent the identification of further sub-fragments within

[3] A fragment of type bond contains at least two children, so we are interested in intersections that contain at least two nodes.

Algorithm 5: Delete Graph

procedure DeleteGraph(GraphId mid)
  RPSDAGNodeId $rid \Leftarrow$ GetRoot(mid)
  DeleteFragment(rid)
  $Roots \Leftarrow Roots \setminus \{(mid, rid)\}$
  CleanRPSDAG()

Algorithm 6: Delete Fragment

procedure DeleteFragment(RPSDAGNodeId fid)
  if $|\{(pid, cid, weight) \in RPSDAG \ : \ cid = fid\}| \leq 1$ **then**
    foreach $(pid, cid, weight)$ *in* $RPSDAG$ *where* $pid = fid$ **do**
      DeleteFragment(cid)
      $RPSDAG \Leftarrow RPSDAG \setminus \{(pid, cid, weight)\}$
    $Codes \Leftarrow \{(code, id, size, type) \in Codes \ : \ id \neq fid\}$

their parents, as new nodes are inserted into the RPSDAG. For example, let us consider the RPSDAG that we created from the two models in Fig. 1. As pointed out before, this RPSDAG contains an edge between B12 and B19 (the latter being a sub-bond of the former). Now, let us assume we remove the model in Fig. 1b from our repository. Since B19 has two parents, namely B12 in model a and P20 in model b, B19 is not deleted by procedure DeleteGraph. However, this node was a sub-fragment which is now left with a single parent of the same type (i.e. a bond). Thus, we also need to remove this sub-fragment from the RPSDAG. In doing so, we let its only parent adopt this node's children, P6 and P7 in our example. In other words, we revert the effects of the adoption that we did when creating a sub-polygon or sub-bond. After this operation, B12 has again its three original children: P6, P7 and P11. Any combination of these children may later be used to create new sub-fragments as new nodes are inserted

Algorithm 7: Clean RPSDAG

procedure CleanRPSDAG()
  foreach $(pid, cid, weight)$ *in* $RPSDAG$ *where* GetType(cid) $\in \{$*"sp", "sb"*$\}$ *and* $\{(pid_2, cid, weight_2) \in RPSDAG : pid_2 \neq pid\} = \varnothing$ **do**
    if GetType(cid) = GetType(pid) **then**
      foreach $ccid \in$ GetChildrenIds(cid) **do**
        $weight =$ GetWeight($cid, ccid$)
        $RPSDAG \Leftarrow$
        $RPSDAG \setminus \{(cid, ccid, weight)\} \cup \{(pid, ccid, weight)\}$
    else
      RestoreType(cid)

into the RPSDAG.

There is also another case which requires cleaning. If a sub-fragment is left with a single parent after DeleteGraph, but the parent's type is different than that of the sub-fragment, it means this sub-fragment corresponds to an original polygon (bond) which has been retyped as a sub-polygon (sub-bond) after procedure InsertSubPolygon (InsertSubBond). In this case CleanRSPDAG invokes function RestoreType() to restore the original type of this fragment (e.g. if the type is "sp" it is changed to "p"). Consider again the example of B19 and now assume we remove the model in Fig. 1a from our repository. After this, B19 is left with one parent only, namely P20 which is a polygon. Thus, B19's type is changed back to "b".

3.4. Complexity

The deletion algorithm performs a depth-first search, which is linear on the size of the graph being deleted. Similarly, the insertion algorithm traverses the inserted graph in linear time. Then, for each fragment to be inserted, it computes its code. Computing the code is linear on the size of the fragment for bonds and polygons, while for rigids it is factorial on the largest number of vertices inside the rigid that share identical labels, as discussed in Section 2.2.[4] If the fragment is a polygon, we compute all LCSs between this polygon and all other polygons in the RPSDAG. Using a suffix tree, this operation is linear on the sum of the lengths of all polygons' canonical codes. If the fragment is a bond, we compute all non-empty intersections between the children of this bond and those of all other bonds. Using a hash table, this operation is linear on the sum of the number of children of all bonds.

3.5. Correctness and completeness

The correctness of the proposed method for clone detection follows from the following observations:

- A node in the RPSDAG corresponds either to a node in the RPST of an indexed model, or to a sub-polygon or a sub-bond.
- The RPST decomposition separates the input process graph into SESE fragments that are either disjoint or have a containment relationship.
- Sub-polygons and sub-bonds are SESE fragments and the insertion algorithm ensures that the sub-polygons (resp. sub-bonds) that appear under a given polygon (bond) are disjoint and are contained by their parent node in the RPSDAG (i.e. the parent polygon or bond).

These observations imply that every node in the RPSDAG is a SESE fragment and that every pair of model fragments indexed in the RPSDAG are either disjoint or in a containment relation. Moreover, we only detect maximal clones, meaning that if a clone C is contained in a clone C', then C' is not returned as a clone. Hence, the set of identified clones are disjoint SESE fragments.

[4] A tighter complexity bound for this problem is given in [12].

The completeness of the proposed method for clone detection stems from the following observations:

- The RPST decomposition can be constructed for any process model [7].[5]
- The RPST decomposition contains one node per SESE subgraph in the original graph, except for SESE subgraphs of type polygon that are directly contained inside another polygon (i.e. sub-polygons) and SESE regions of type bond directly contained inside another bond (i.e. sub-bonds). The RPSDAG then extracts these sub-polygons and sub-bonds when it is detected that two indexed polygons (bonds) share a common sub-polygon (sub-bond). Sub-polygons are detected using a longest common substring algorithm, thus ensuring that maximal-sized shared sub-polygons are identified.

3.6. Fragment query

The proposed index can also be used to identify all graphs in a repository that contain a query graph. By building the RPSDAG first, we can perform this operation efficiently as opposed to using subgraph isomorphism check, which is NP-complete [13]. In other words, we shift the bulk of the computational complexity from run-time (query execution) to design-time (repository creation).

Algorithm 8: Fragment Query

```
procedure FragmentQuery(RPSDAGNodeId id) returns
{RPSDAGNodeId, RPSDAGNodeId}
    {RPSDAGNodeId, RPSDAGNodeId} MP ⇐ ∅
    foreach (pid, cid, weight) in RPSDAG where cid = id do
        R ⇐ GetRootIds(pid)
        foreach rid in R do
            MP ⇐ MP ∪ {(GetGraphId(rid), pid)}
    if MP = ∅ then
        MP ⇐ {(GetGraphId(id), id)}
    return MP
```

The execution of a query (cf. Algorithm 8) takes a fragment id as input, and returns the id of all models the query fragment occurs in, and for each model, also the id of the parent fragment containing the query fragment. In this way we can locate the query fragment exactly within each model satisfying the query. To do so, we first retrieve the root fragment id of all graphs containing the input fragment by traversing the RPSDAG upwards from the input fragment (function GetRootIds()). Then for each root id we retrieve the corresponding model id from table Roots using function GetGraphId(), which returns 0 if the graph does not exist. Algorithm 8 assumes the id of the query fragment is known. Otherwise, we can retrieve it from table Codes.

[5]Specifically, [7] shows that an RPST can be constructed for any arbitrary directed graph such that every node is on a path from a source node to a sink node, which is a basic property of process models.

Algorithm 9: GetRootIds

```
procedure GetRootIds(RPSDAGNodeId id) returns {RPSDAGNodeId}
    {RPSDAGNodeId} R ⇐ ∅
    foreach (pid, cid, weight) in RPSDAG where cid = id do
        R ⇐ R ∪ GetRootIds(pid)
    if R = ∅ then
        R ⇐ {id}
    return R
```

4. Evaluation

This section reports on a series of tests to evaluate the performance of the RPSDAG as well as the usefulness of clone detection in practical settings.

4.1. Evaluation dataset

We evaluated the RPSDAG using four datasets:

- The collection of SAP R3 reference process models [14]
- A model repository obtained from Suncorp, Australia's largest insurance company.
- Two collections from the IBM BIT process library [15], namely collections A and B3. In the BIT process library there are 5 collections (A, B1, B2, B3 and C). We excluded collections B1 and B2 because they are earlier versions of B3, and collection C because it is a mix of models from different sources and as such it does not contain any clones.

The SAP repository contains 595 models with sizes ranging from 5 to 119 nodes (average 22.28, median 17). The insurance repository contains 363 models ranging from 4 to 461 nodes (average 27.12, median 19). The BIT collection A contains 269 models ranging from 5 to 47 nodes (average 17.01, median 16) while collection B3 contains 247 models with 5 to 42 nodes (average 12.94, median 11). The examples in Figure 1 are extracts of the insurance models with node labels altered to protect confidentiality.

4.2. Performance evaluation.

We first evaluated the insertion times. Obviously inserting a new model into a nearly-empty RPSDAG is less costly than doing so in an already populated one. To factor out this effect, we randomly split each dataset as follows: One third of the models were used to construct an initial RPSDAG and the other two-thirds were used to measure insertion times. In the SAP repository, 200 models were used to construct an initial RPSDAG. Constructing the initial RPSDAG took 26s. In the insurance company repository, the initial RPSDAG contained 121 models and its construction took 26s. For the BIT collections A and B3, 90 and 82 models respectively were used for

constructing the initial RPSDAG. Constructing the initial RPSDAGs took 8.6s for collection A and 6.1s for collection B3. All tests were conducted on a PC with a dual core Intel processor, 1.8 GHz, 4 GB memory, running Microsoft Windows 7 and Oracle Java Virtual Machine v1.6. The RPSDAG was implemented as a Java console application on top of MySQL 5.1. Each test was run 10 times and the execution times obtained across the ten runs were averaged.

Table 1 summarizes the insertion time per model for each collection (min, max, avg., std. dev., and 90^{th} percentile). All values are in milliseconds. These statistics are given for three cases: without sub-polygon nor sub-bond clone detection, with sub-polygon clone detection, and with both sub-polygon and sub-bond clone detection. Herewith, we use the term *sub-fragment clone detection* to refer to both sub-polygon and sub-bond clone detection collectively.

We observe that the average insertion time per model is 3-5 times larger when sub-polygon is performed. This overhead comes from the step where we compare a polygon being inserted with each already-indexed polygon and we compute the longest-common substring of their canonical codes. As mentioned earlier, this operation could be optimized at the implementation level by storing all the indexed polygons in a suffix tree [10]. Sub-bond detection also introduces a visible overhead because of the step where an inserted bond is compared against each indexed bond. This latter step could be optimized by using bitsets to represent the set of children of each bond.

Despite the fact that the tool implementation does not incorporate these optimizations, the average execution times remain in the order of tens to hundreds of milliseconds across all model collections even with sub-poloygon and sub-bond detection. The highest average insertion time (295ms) is observed for the Insurance collection. This collection contains some models with large rigid components in their RPST. In particular, one model contained a rigid component in which two task labels appeared nine times each – i.e. nine tasks had one of these labels and nine tasks had the other – and these tasks were all preceded by the same gateway. Putting aside this extreme case, all insertion times were under one second and in 90% of the cases (l_{90}), the insertion times in this collection were under 50ms without sub-fragment detection and 400ms with sub-fragment detection. Thus we can conclude that the proposed technique scales up to real-sized model collections.

	min	max	avg	std	l_{90}	
SAP	4	85	20	14	40	no sub-fragments
Insurance	5	1722	32	113	49	
BIT A	3	58	12	9	23	
BIT B3	5	467	14	9	28	
SAP	4	482	97	81	202	sub-polygons only
Insurance	26	4402	126	291	222	
BIT A	18	128	41	20	59	
BIT B3	5	150	33	24	69	
SAP	10	859	232	124	398	sub-bonds + sub-polygons
Insurance	81	5043	295	365	390	
BIT A	71	226	129	40	187	
BIT B3	15	192	75	33	115	

Table 1: Model insertion times (in ms).

As explained in Section 3, once the models are inserted, we can find all clones with

a SQL query. The query to find all clones with at least two nodes and occurring at least twice from the SAP reference models takes 75ms on average, 90ms for the insurance models, 118 and 116ms for the BIT collections A and B3.

Finally, we evaluated the performance of the RPSDAG for handling fragment queries. To this end, we randomly selected model fragments of sizes from five to 15 nodes from the SAP repository. In other words, for each fragment size (from 5 to 15), we randomly selected a fragment among all fragments of this size in the repository. For each fragment, we ran a query to retrieve all occurrences of this fragment in the collection of models. Each query was executed five times and the execution times were averaged. The recorded average execution times ranged from 15 to 35 ms, with an average of 24 ms and st. dev. of 6 ms.

4.3. Refactoring gain.

One of the main applications of clone detection is to refactor the identified clones as shared subprocesses in order to reduce the size of the model collection and increase its navigability. In order to assess the benefit of refactoring clones into subprocesses, we define the following measure.

Definition 4.1. *The* refactoring gain *of a clone is the reduction in number of nodes obtained by encapsulating that clone into a separate subprocess, and replacing every occurrence of the clone with a task that invokes this subprocess. Specifically, let S be the size of a clone, and N the number of occurrences of this clone.*[6] *Since all occurrences of a clone are replaced by a single occurrence plus N subprocess invocations, the refactoring gain is: $S \cdot N - S - N$.*

Given a collection of models, the total refactoring gain *is the sum of the refactoring gains of the clones of non-trivial clones (size ≥ 2) in the collection.*

It should be noted that when a sub-bond is refactored out, we need to introduce an additional split and a join in the refactored sub-bond in order to separate it from the parent bond. Accordingly, in the case of sub-bonds, the refactoring gain is defined as: $S \cdot N - S - N - 2$.

Table 2 summarizes the total refactoring gain for each model collection. The first two columns correspond to the total number of clones detected and the total refactoring gain without any sub-fragment refactoring. The third and fourth column show number of clones and refactoring gain with sub-polygon refactoring. Finally, the last two columns give the number of clones and gain attained with both sub-polygon and sub-bond refactoring. The table shows that a significant number of clones can be found in all model collections and that the size of these model collections could be reduced by 8.5-17% if clones were factored out into shared subprocesses. The table also shows that sub-fragment clone detection adds significant value to the clone detection method. For instance, in the case of the insurance models, we obtain over twice more clones and twice more refactoring gain when sub-fragment clone detection is performed. The results demonstrate the potential usefulness of detecting and refactoring both sub-polygons and sub-bonds.

[6] If a clone appears multiple times under the same parent node in the RPSDAG (sibling clones), this should be counted as multiple occurrences.

Sibling clones – i.e. identical fragments appearing under the same parent node in the RPSDAG – represented only a negligible fraction of all clones (7 sibling clones in the SAP repository, 1 in the Insurance repository, none in the IBM repositories).

	No sub-fragments		With sub-polygons		With sub-polygons + sub-bonds	
	Nr. clones	gain	Nr. clones	gain	Nr. clones	gain
SAP	304	1359 (10.3%)	490	1859 (14%)	563	2261 (17.1%)
Insurance	108	395 (4%)	280	884 (9%)	302	933 (9.5%)
BIT A	57	195 (4.3%)	174	384 (8.4%)	178	391 (8.5%)
BIT B3	19	208 (6.5%)	49	259 (6.6%)	61	293 (9.2%)

Table 2: Total refactoring gain without and with sub-fragment refactoring.

Table 3 shows more detailed statistics of the clones found with sub-fragment detection (including both sub-polygons and sub-bonds). The first three columns give statistics about the sizes of the clones found, the next three columns refer to the frequency (number of occurrences) of the clones, and the last three correspond to the efactoring gain. We observe that while the average clone size is relatively small (three-five nodes), there are also large clones with sizes of 30+ nodes.

	Size			# occurrences			Refactoring gain		
	avg	max	std. dev.	avg	max	std. dev.	avg	max	std. dev.
SAP	5.01	41	4.02	2.37	11	0.89	4.02	44	5.82
Insurance	3.73	32	3.20	2.71	41	3.05	3.09	79	7.46
BIT A	3.02	16	2.10	2.74	9	1.28	2.20	15	3.10
BIT B3	3.36	9	1.51	3.69	20	3.39	4.80	37	8.09

Table 3: Statistics of detected clones (with sub-fragment detection).

It might be desirable not to refactor out small clones, as it would add complexity and fragmentation in the model collection by introducing many "small" subprocesses and making the dataset more difficult to navigate.

The smallest process model in the evaluated datasets has 4 nodes. Thus, it would make sense to refactor out only clones that have at least 4 nodes in order not to introduce subprocesses that are smaller than those that process modelers would normally define themselves. Table 4 shows the refactoring gains obtained if we only consider clones with at least four nodes. We observe that refactoring out only clones of at least four nodes still gives us a significant amount of refactoring gain. For the SAP collection we still identify 293 clones out of 555 clones (with sub-fragment detection), leading to a refactoring gain of 14.7% (versus 17.1% if we consider all clones).

	No. Clones	Refactoring gain
SAP	300	1949 (14.7%)
Insurance	107	613 (6.25%)
BIT A	39	234 (5.11%)
BIT B3	27	171 (5.36%)

Table 4: Refactoring clones with size of at least 4 nodes (with sub-fragment detection).

5. Related Work

This article is an extended version of a previous conference paper [16]. The extensions include the ability to detect sub-bond clones and sibling clones, as well as the application of the RPSDAG to handle fragment queries. The experimental evaluation was extended to assess the performance of the extended RPSDAG and of the query processing, and to assess the usefulness of sub-bond and sibling clone detection. Below, we review related work along the following areas: i) clone detection in software repositories; ii) clone detection in model-drive engineering; iii) graph database indexing; iv) refactoring techniques for process model repositories; and v) query languages for process model repositories.

Clone detection in software repositories

Clone detection in software repositories has been an active field for several years. According to [17], approaches can be classified into: textual comparison, token comparison, metric comparison, abstract syntax tree (AST) comparison, and program dependence graphs (PDG) comparison. The latter two categories are close to our problem, as they use a graph-based representation. In [18], the authors describe a method for clone detection based on ASTs. The method applies a hash function to subtrees of the AST in order to distribute subtrees across buckets. Subtrees in the same bucket are compared by testing for tree isomorphism. This work differs from ours in that RPSTs are not perfect trees. Instead, RPSTs contain rigid components that are irreducible and need to be treated as subgraphs—thus tree isomorphism is not directly applicable.

Code clone detection using PDGs has been investigated in [19]. The PDG is a directed graph where nodes correspond to lexer tokens, and edges correspond to control, data and reference dependencies. A subgraph isomorphism algorithm is used for clone detection. This technique is unsuitable for online processing due to performance and memory requirements [17]. In contrast, we employ canonical codes instead of pairwise subgraph isomorphism detection. The bottleneck is that we have to potentially consider all permutation of gateways in a rigid component in order to construct the canonical code. Our experiments show however that this can be achieved in sub-second times even for large process models. Another difference between our approach and those based on PDG is that we take advantage of the RPST in order to decompose the process graph into SESE fragments, allowing us to focus on smaller fragments at once.

Clone detection in model-driven engineering

Work on clone detection has also been undertaken in the field of model-driven engineering. [20] describes 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. Again, the main difference with our work is that we use canonical codes instead of subgraph isomorphism detection.

In [21], the authors describe two methods for exact and approximate matching of clones for Simulink models. In the first method, they apply an incremental, heuristic subgraph matching algorithm. In the second approach, graphs are represented by a set of vectors built from graph features: e.g. path lengths, vertex in/out degrees, etc.

An empirical study shows that this feature-based approximate matching approach improves pre-processing and running times, while keeping a high precision. However, this data structure does not support incremental insertions/deletions.

Graph database indexes

Our work is also related to graph database indexing and is inspired by [9]. Other related indexing techniques for graph databases include GraphGrep [13] – an index designed to retrieve paths in a graph that match a regular expression. This problem however is different from that of clone detection since clones are not paths (except for polygons). Another related index is the closure-tree [22]. Given a graph G, the closure-tree can be used to retrieve all indexed graphs in which G occurs as a subgraph. We could use the closure tree to index a collection of process graphs so that when a new graph is inserted we can check if any of its SESE regions appears in an already indexed graph. However, the closure tree does not directly retrieve the exact set of graphs where a given subgraph occurs. Instead, it retrieves a "candidate set" of graphs. An exact subgraph isomorphism test is then performed against each graph in the candidate set. In contrast, by storing the canonical code of each SESE region, the RPSDAG obviates the need for this subgraph isomorphism testing.

In [23], we described an index to retrieve process models in a repository that exactly or approximately match a given model fragment. In this approach, process models are represented as Petri nets and paths in the process models are used as index features. Given a collection of models, a B+ tree is used to reduce the search space by discarding those models that do not contain any path of the query model. The remaining models are tested for subgraph isomorphism.

Refactoring process model repositories

In [24], eleven process model refactoring techniques, called "smells", 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". Recent work has shown that several other types of refactoring opportunities can be semi-automatically identified by computing similarity metrics on model fragments [25]. Specifically, this related work suggests to identify refactoring opportunities by computing similarity metrics on every pair of (SESE) fragments within a given repository of process models. This approach can be used in particular to identify "subprocess extraction" opportunities, including subprocess extraction opportunities where the refactored fragments are not identical. However, in doing so, the approach leads to many "false positives", and thus requires additional filters or fine-tuning. Our approach is less general, but does not produce false positives: Every clone detected by our technique constitutes a subprocess extraction opportunity. Also, our approach explicitly aims to reduce computational overhead, while this is not a concern in the work reported in [25]. In other words, our technique strikes different tradeoffs and is potentially complementary to the techniques presented in [25].

Querying process model repositories

In this paper we showed how the RPSDAG can be used to efficiently query a repository for the existence of a given process fragment. Two major research efforts have been dedicated to the development of query languages for process model repositories: BP-QL and BPMN-Q. BP-QL [26] is a graphical query language based on an abstract representation of BPEL, which is supported by a formal model of graph grammars for query processing. BP-QL can be used to query process specifications written in BPEL rather than possible executions, and ignores the run-time semantics of certain BPEL constructs such as conditional execution and parallel execution.

BPMN-Q [27, 28] is also a visual query language which extends a subset of the BPMN modelling notation and supports graph-based query processing. Similarly to BP-QL, BPMN-Q captures the structural relationships between tasks. In [29], the authors explore the use of an information retrieval technique to derive similarities of activity names, and develop an ontological expansion of BPMN-Q to tackle the problem of querying business processes that are developed with different terminologies.

The above works do not focus on efficient querying but rather on how to formulate process model queries graphically, and on how to parse these queries. As such, these works complement our work on fragment querying by providing an interface through which users can submit queries.

6. Conclusion

We presented a technique to index process models in order to identify duplicate SESE fragments (clones) that can be refactored into shared subprocesses. The proposed index, namely the RPSDAG, combines a method for decomposing process models into SESE fragments (the RPST decomposition) with a method for generating a unique string to identify a labeled graph (canonical codes). Canonical codes are used to determine whether a SESE fragment in a model appears elsewhere in the same or in another model.

The RPSDAG has been implemented and tested using process model repositories from industrial practice. In addition to demonstrating the scalability of the RPSDAG, the experimental results show that a significant number of non-trivial clones can be found in industrial process model repositories. In one repository, more than 560 non-trivial clones were found. By refactoring these clones, the overall size of the repository is reduced by around 17%, which arguably would enhance the repository's maintainability. We also showed that the RPSDAG can be used to efficiently answer "fragment queries", that is, retrieving all occurrences of a given model fragment in a repository.

In separate work [30], we adapted the RPSDAG to deal with concurrent editing and change propagation in the context of repositories of versioned process models. Concurrent editing is handled by allowing modelers to obtain locks at the level of individual fragments of a process model as opposed to locking an entire model. This operation corresponds to placing a lock on a node of the RPSDAG. Change propagation is achieved by allowing modelers to determine, on a fragment-by-fragment basis, whether or not changes made in a version of a model should be propagated to other versions.

A standalone release of the RPSDAG implementation, together with sample models, is available at: `http://apromore.org/tools`. The RPSDAG implementation can be optimized in several ways: (i) by using suffix trees for identifying the longest common substrings between the code of an inserted polygon and those of already-indexed polygons; (ii) by storing the canonical codes of each fragment in a hash index in order to speed up retrieval; (iii) by using the Nauty library for computing canonical codes (`http://cs.anu.edu.au/~bdm/nauty`). Nauty implements several optimizations that could complement those described in Section 2.2.

One issue that arises when extracting clones into shared subprocesses is that of giving meaningful labels to the subprocesses. To assist analysts in this task, it may be desirable to automatically suggest possible labels for the sub-processes to be extracted. Such suggestions could be derived by analyzing the labels of the tasks inside the clone and using meronymy relations to compute aggregate labels as investigated in [31].

Another avenue for future work is to extend the proposed technique in order to identify *approximate* clones. This has applications in the context of process standardization, when analysts seek to identify similar but non-identical fragments and to replace them with standardized fragments in order to increase the homogeneity of work practices.

Acknowledgments This research is funded by the Estonian Science Foundation and ERDF via the Estonian Centre of Excellence in Computer Science (Uba, Dumas and García-Bañuelos), and by ARC Linkage Grant "LP110100252" and NICTA Queensland Lab (La Rosa).

References

[1] M. Rosemann, Potential pitfalls of process modeling: part A, Business Process Management Journal 12 (2) (2006) 249–254.

[2] H. A. Reijers, R. S. Mans, R. A. van der Toorn, Improved model management with aggregated business process models, Data Knowl. Eng. 68 (2) (2009) 221–243.

[3] M. La Rosa, M. Dumas, R. Dijkman, Business Process Model Merging: An Approach to Business Process Consolidation, ACM Transactions on Software Engineering and Methodology (2012) (to appear).

[4] R. Dijkman, M. La Rosa, H. Reijers, Managing Large Collections of Business Process Models – Current Techniques and Challenges, Computers in Industry 63 (2) (2012) 91–97.

[5] R. Koschke, Identifying and Removing Software Clones, in: Software Evolution, Springer, 2008, pp. 15–36.

[6] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, Data Knowl. Eng. 68 (9) (2009) 793–818.

[7] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified Computation and Generalization of the Refined Process Structure Tree, in: Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September

16-17, 2010. Revised Selected Papers, Vol. 6551 of Lecture Notes in Computer Science, Springer, 2010, pp. 25–41.

[8] M. La Rosa, H. Reijers, W. van der Aalst, R. Dijkman, J. Mendling, M. Dumas, L. Garcia-Banuelos, APROMORE: An Advanced Process Model Repository, Expert Systems with Applications 38 (6) (2011) 7029–7040.

[9] D. W. Williams, J. Huan, W. Wang, Graph Database Indexing Using Structured Graph Decomposition, in: Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey, IEEE Computer Society, 2007, pp. 976–985.

[10] E. Ukkonen, On-Line Construction of Suffix Trees, Algorithmica 14 (3) (1995) 249–260.

[11] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[12] L. Babai, Monte-Carlo algorithms in graph isomorphism testing, Tech. Rep. D.M.S. No. 79-10, Université de Montréal, Dép. de mathématiques et de statistique (1979).

[13] D. Shasha, J. T.-L. Wang, R. Giugno, Algorithmics and Applications of Tree and Graph Searching, in: Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA, ACM, 2002, pp. 39–52.

[14] G. Keller, T. Teufel, SAP R/3 Process Oriented Implementation: Iterative Process Prototyping, Addison-Wesley, 1998.

[15] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Instantaneous Soundness Checking of Industrial Business Process Models, in: Business Process Management, Vol. 5701 of Lecture Notes in Computer Science, Springer, 2009, pp. 278–293.

[16] R. Uba, M. Dumas, L. García-Bañuelos, M. L. Rosa, Clone Detection in Repositories of Business Process Models, in: Proceedings of the 9th International Conference on Business Process Management (BPM 2011), Vol. 6896 of Lecture Notes in Computer Science, Springer, Clermont-Ferrand, France, 2011, pp. 248–264.

[17] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and Evaluation of Clone Detection Tools, IEEE Trans. Software Eng. 33 (9) (2007) 577–591.

[18] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, L. Bier, Clone Detection Using Abstract Syntax Trees, in: Proceedings of the International Conference on Software Maintenance (ICSM'98), 16-19 November, 1998, Bethesda, Maryland, USA, IEEE Computer Society, 1998, pp. 368–377.

[19] J. Krinke, Identifying Similar Code with Program Dependence Graphs, in: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), 2-5 October 2001, Suttgart, Germany, IEEE Computer Society, 2001, pp. 301–309.

[20] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, S. Teuchert, Clone Detection in Automotive Model-Based Development, in: Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany, 7.-9. April 2008, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme, TU Braunschweig, Institut für Software Systems Engineering, 2008, pp. 57–67.

[21] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, T. N. Nguyen, Complete and accurate clone detection in graph-based models, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, IEEE Computer Society, 2009, pp. 276–286.

[22] H. He, A. K. Singh, Closure-Tree: An Index Structure for Graph Queries, in: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, IEEE Computer Society, 2006, p. 38.

[23] T. Jin, J. Wang, N. Wu, M. L. Rosa, A. H. M. ter Hofstede, Efficient and Accurate Retrieval of Business Process Models through Indexing - (Short Paper), in: On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I, Vol. 6426 of Lecture Notes in Computer Science, Springer, 2010, pp. 402–409.

[24] B. Weber, M. Reichert, J. Mendling, H. Reijers, Refactoring large process model repositories, Computers in Industry 62 (5) (2011) 467–486.

[25] R. M. Dijkman, B. Gfeller, J. M. Küster, H. Völzer, Identifying refactoring opportunities in process model repositories, Information & Software Technology 53 (9) (2011) 937–948.

[26] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL, Inf. Syst. 33 (6) (2008) 477–507.

[27] A. Awad, BPMN-Q: A language to query business processes, in: M. Reichert, S. Strecker, K. Turowski (Eds.), Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA'07), St. Goar, Germany, October 8-9, 2007, Vol. P-119 of LNI, GI, 2007, pp. 115–128.

[28] A. Awad, G. Decker, M. Weske, Efficient compliance checking using BPMN-Q and temporal logic, in: M. Dumas, M. Reichert, M.-C. Shan (Eds.), Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings, Vol. 5240 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 326–341.

[29] A. Awad, A. Polyvyanyy, M. Weske, Semantic querying of business process models, in: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, September 15-19, 2008, Munich, Germany, IEEE Computer Society, 2008, pp. 85–94.

[30] C. Ekanayake, M. L. Rosa, A. ter Hofstede, M.-C. Fauvet, Fragment-Based Version Management for Repositories of Business Process Models, in: CoopIS, Vol. 7044 of Lecture Notes in Computer Science, Springer, 2011, pp. 20–37.

[31] S. Smirnov, R. M. Dijkman, J. Mendling, M. Weske, Meronymy-based aggregation of activities in business process models, in: Proceedings of the 29th International Conference on Conceptual Modeling (ER 2010), Vancouver, Canada, Vol. 6412 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–14.