HAL
open science

Fragment-based Version Management for Repositories of Business Process Models

Chathura C. Ekanayake, Marcello La Rosa, Arthur H. M. ter Hofstede, Marie-Christine Fauvet

To cite this version:

Chathura C. Ekanayake, Marcello La Rosa, Arthur H. M. ter Hofstede, Marie-Christine Fauvet. Fragment-based Version Management for Repositories of Business Process Models. OTM 2011 - Confederated international conference on On the Move to Meaningful Internet Systems, Oct 2011, Crete, Greece. pp.20-37, 10.1007/978-3-642-25109-2_3 . hal-00763160

HAL Id: hal-00763160

https://hal.science/hal-00763160

Submitted on 10 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Queensland University of Technology
Brisbane Australia

This is the author's version of a work that was submitted/accepted for publication in the following source:

Ekanayake, Chathura C., La Rosa, Marcello, ter Hofstede, Arthur H.M., & Fauvet, Marie-Christine (2011) *Fragment-based version management for repositories of business process models.* (Unpublished)

This file was downloaded from: http://eprints.qut.edu.au/41538/

© Copyright 2011 The Authors

Notice: *Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source:*

Fragment-based Version Management for Repositories of Business Process Models

C.C. Ekanayake[1], M. La Rosa[1], A.H.M. ter Hofstede[1,2], and M.-C. Fauvet[3]

[1] Queensland University of Technology, Australia
chathura.ekanayake@student.qut.edu.au,{m.larosa,a.terhofstede}@qut.edu.au
[2] Eindhoven University of Technology, The Netherlands
[3] University of Grenoble, France
marie-christine.fauvet@imag.fr

Abstract. As organizations reach higher levels of Business Process Management maturity, they tend to accumulate large collections of process models. These repositories may contain thousands of activities and be managed by different stakeholders with varying skills and responsibilities. However, while being of great value, these repositories induce high management costs. Thus, it becomes essential to keep track of the various model versions as they may mutually overlap, supersede one another and evolve over time. We propose an innovative versioning model, and associated storage structure, specifically designed to maximize sharing across process models and process model versions, reduce conflicts in concurrent edits and automatically handle controlled change propagation. The focal point of this technique is to version single process model fragments, rather than entire process models. Indeed empirical evidence shows that real-life process model repositories have numerous duplicate fragments. Experiments on two industrial datasets confirm the usefulness of our technique.

1 Introduction

Organizations need to develop process models to document different aspects of their business operations. For example, process models are used to communicate changes in existing operations to relevant stakeholders, document procedures for compliance inspection by auditors or guide the development of IT systems [30]. Such process models are constantly updated to suit new or changed requirements, and this typically leads to different versions of the same process model. Thus, organizations tend to accumulate large numbers of process models over time [24]. For example, Suncorp, one of the largest Australian insurers, maintain a repository of 6,000+ process models [23], whereas the Chinese railway company CNR has 200,000+ models.

The requirement to deal with an increasing number of process models within organizations poses a maintenance challenge. Especially, it becomes essential to keep track of the various models as they may mutually overlap, supersede one another and evolve over time. Moreover, process models in large organizations are typically edited by stakeholders with varying skills, responsibilities and goals, sometimes distributed across independent organizational units [6]. This calls for techniques to efficiently store process models and manage their evolution over time.

In this paper, we propose a novel *versioning model* and associated *storage structure* which are specifically designed for process model repositories. The main innovation lies in storing and versioning single process fragments (i.e. subgraphs), rather than entire process models. In this way duplicate fragments across different process models, or across different versions of the same process model, are stored only once. In fact, empirical evidence [38] shows that industrial process model collections feature a high number of duplicate fragments. This occurs as new process models are created by copying fragments from existing models within the same collection. For example, we identified nearly 14% of redundant content in the SAP R/3 reference model [19]. Further, when a new process model version is created, only a subset of all its fragments typically changes, leaving all other fragments unchanged across all versions of the same model.

Besides effectively reducing the storage requirements of (large) process model repositories, our technique provides three benefits. First, it keeps track of shared fragments both *horizontally*, i.e. across different models, and *vertically*, i.e. across different versions of the same model. As a result, this information is readily available to the repository users, who can monitor the various relations among process model versions. Second, it increases *concurrent editing*, since locks can be obtained at the granularity of single fragments. Based on the assumption that different users typically work on different fragments at the same time, it is no longer necessary to lock an entire process model, but only those fragments that will actually be affected by a change. As a result, the use of traditional conflict resolution techniques is limited to situations in which the same fragment is edited by multiple users concurrently. Finally, our technique provides sophisticated *change propagation*. For example, if an error is detected in a shared fragment, the fix can be automatically propagated to all process models containing that fragment, without having to edit each process model individually. This in turn can facilitate reuse and standardization of best business practices throughout the process model repository. To the best of our knowledge, the use of process fragments for version control, concurrency control (i.e. locking) and change propagation of process model collections has not been studied in existing research. Commercial BPM suites only offer propagation of attribute changes at the node level, e.g. a label change.

The proposed technique is independent of the process modeling language being adopted as all the developed methods operate on an abstract modeling notation. Thus, it is possible to manage processes modeled in a variety of languages (e.g. BPEL, YAWL, BPMN, EPC) with our technique. We implemented this technique on top of the MySQL relational DBMS and used the prototype to conduct experiments on two industrial process model collections. The results show that the technique yields a significant gain in storage space and demonstrate the usefulness of its locking and change propagation mechanisms.

We present our technique in three steps. First, we introduce the versioning model in Sec. 2. Next, we describe our locking mechanism in Sec. 3 and finally our controlled changed propagation in Sec. 4. In Sec. 5 we discuss the storage structure used to implement our technique on top of relational DBMSs, while in Sec. 6 we present the algorithms for manipulating this data structure. We report the experimental setup and results in Sec. 7, and discuss related work in Sec. 8. We draw conclusions in Sec. 9.

2 Versioning model

We define process model versions according to a branching model which is inspired by popular version-control systems such as Concurrent Version Systems (CVS) [4] and Apache Subversion (SVN).[4] Accordingly, each process model can have one or more *branches* to account for co-existing developments. Each branch contains a sequence of process versions and has a unique name within a process model.

A new branch can be created by "branching out" from a version in another existing branch, where the existing branch may belong to the same process model (*internal branching*) or to another process model (*external branching*). The *primary* branch is the first branch being created for a process model, and as such it can be new or be derived via external branching. Non-primary branches of a process model can only be derived via internal branching. Only the last version of a branch, namely the *current version* can be modified.

A modification to a current version produces a new version in the same branch which becomes the current version. According to this versioning model, a specific version of a process model is referred to by the tuple (process model name, branch name, version number). Optionally, a version may have a name which needs not be unique. This model is shown in Fig. 1 by using an example from the insurance domain. Here the primary branch is new and named "Home", whereas "Motor", "Private" and "Commercial" are all secondary branches. For example, version 1.0 of the Motor branch, named "alpha", is derived from version 1.1 of the Home branch, named "signed".

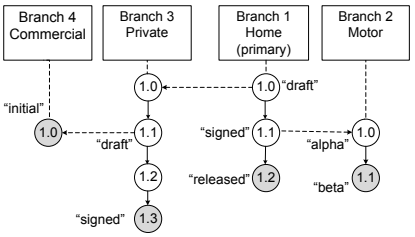

Fig. 1. Process model versioning (current version of each branch is shaded).

The focal idea of our versioning model is to use process model fragments as storage units. To obtain all fragments from a process model, we use the Refined Process Structure Tree (RPST) [39]. The RPST is a linear-time method to decompose a process model into a tree of hierarchical *SESE fragments*. A SESE fragment is a subgraph of a process model with a single entry and a single exit node. Each fragment in the hierarchy contains all fragments at the lower level, but fragments at the same level are disjoint. Thus, a given process model has only one RPST decomposition. The advantage of using SESE fragments is that they are modular: any change inside a fragment does not affect other fragments outside the modified fragment. Fig. 2 shows version 1.0 of the Home insurance claims process model, and its RPST decomposition. The notation is BPMN.

For each model, we store its SESE fragments with their composition relationships. A fragment may contain one or more child fragments, each of which may also contain child fragments, forming a tree structure. Fig. 3 shows the fragment version tree of the process model in Fig. 2.

[4] `http://subversion.apache.org`

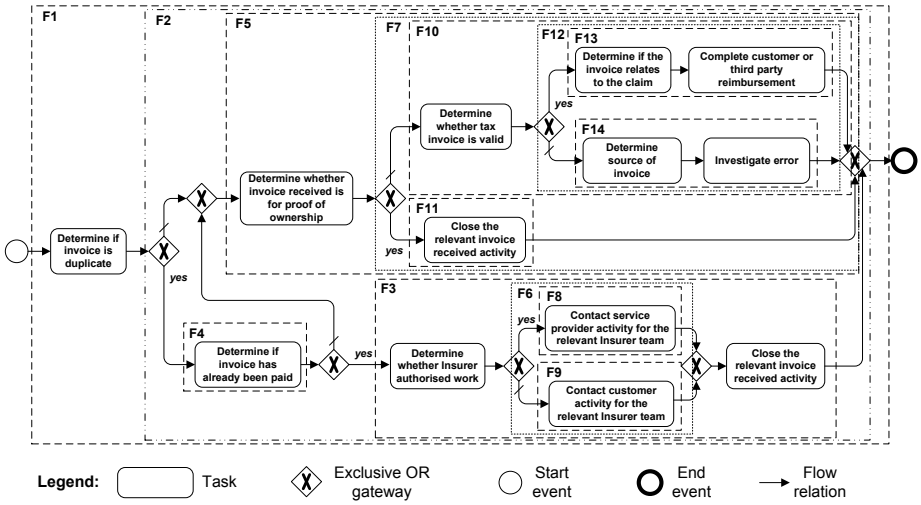

Fig. 2. Version 1.0 of the Home insurance claims process model, and its RPST fragments.

We maintain a version history for each fragment. Each fragment has a sequence of versions and the latest version is named as the *current* version. When a new fragment is added, its version sequence starts with 1 and is incremented by one for each subsequent version. Fig. 3 depicts fragments as rectangles and fragment versions as circles; version numbers are shown inside circles. As all fragments in this example are new, each fragment has version 1. Each process model version points to the root fragment version of its fragment version tree.

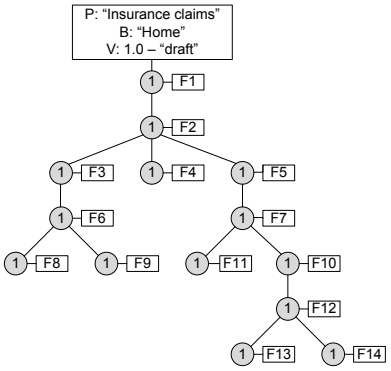

Fig. 3. RPST of model in Fig. 2.

By using fragments as units of storage, we can efficiently support version control, change management and concurrency control for process models. Before describing how we realize such operations, we explain how a fragment is stored in the repository. Each fragment version needs to store its *composition relationships* and its *structure*.

The composition relationships contain the identifiers of all the immediate child fragment versions. The structure of a fragment version is the subgraph of that fragment version where the subgraphs of all its child fragment versions are replaced by placeholders called *pockets*. Each pocket is associated with an identifier and within the structure of a particular fragment version, it points to one child fragment version. In this way we can maximize reuse across fragments, since two fragments can

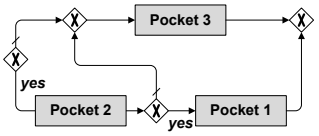

Fig. 4. Structure of fragment F2 from the model in Fig. 2.

share the same structure but point to different child fragment versions from within their pockets. Fig. 4 shows the structure of fragment F2 from Fig. 2. This structure contains three child fragments, each represented by a pocket. In the case of version 1 of F2, pocket 1 points to version 1 of F3, pocket 2 to version 1 of F4 and pocket 5 to version 1 of F5. Next, we describe how to reuse structures by mapping different child fragment versions to pockets.

2.1 Vertical sharing

Process models are not static artifacts but evolve with an organization. As we store individual fragments, all unmodified fragments can be shared across different versions of the same process model. We call this *vertical sharing*. When a new version of a process model is created, only those fragments that have changed or that have been added are stored. Fig. 5 shows the derivation of version 1.1 from version 1.0 of the Home insurance claims process by modifying fragment F3. Fragment F3 is modified by removing F6 and adding F25 and F32. This leads to a new version of F3 with the modified content (version 2). In addition, new versions of F2 and F1 need to be created with the modified composition relationships. All other fragments (i.e. F4 to F14) remain the same and are shared between version 1.0 and 1.1 of the Home insurance process.

As we mentioned earlier, we reuse structures of fragments across subsequent fragment versions in order to avoid redundancy. For example, changing fragment F3 does not affect the structure of fragment F2. However, a new version of F2 has to be created to represent the modified composition relationships (i.e. replacement of version 1 of F3 with version 2). Thus, the structure can be shared across versions 1 and 2 of F2. Let us consider the structure of version 1 of F2 as shown in Fig. 4. According to the example, version 1 of F2 maps version 1 of fragments F3, F4 and F5 to pockets 1, 2 and 3 respectively. In version 2 of F2, the structure does not change except for the mapping of pocket 1 which now points to version 2 of F3. Thus, we reuse the structure of version 1 of F2 in its version 2 simply by changing the mapping of its pocket 1.

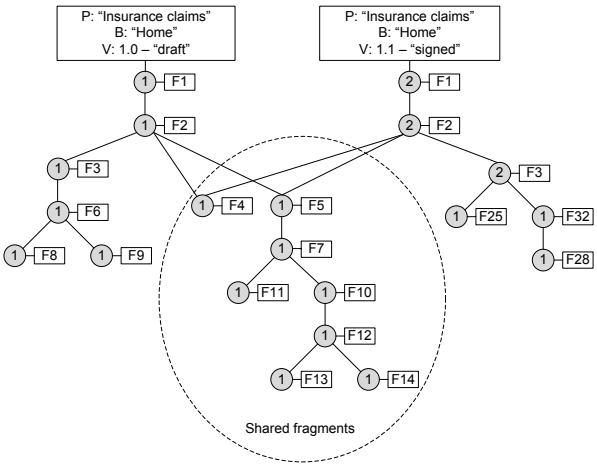

Fig. 5. Sharing fragments across multiple versions of the same process model.

2.2 Horizontal sharing

Real-life process model repositories hardly have unique process models. It is common in fact that multiple process models share common fragments. For example, we identi-

fied 840 duplicate fragments in the SAP reference model. In order to avoid such redundancy, we also allow fragment versions to be shared among multiple branches within or across process models. We call this *horizontal sharing*. By keeping track of such derivation relationships, we can efficiently propagate changes and keep the repository in a consistent state. As an example, Fig. 6 shows the relationship between version 1.2 of the Home insurance branch and version 1.1 of the Motor insurance branch, which share fragments F3 and F5, and their child fragments. Similar sharing relations can exist between branches of different process models.

3 Locking

If two or more users try to modify two overlapping sections within the same process model or across different process models, the resulting process model(s) may become inconsistent. The solution used by current process model repositories to avoid such conflicts is to lock an entire process model before editing it. However, such a solution limits the ability for collaboration, especially in light of the current trend for collaborative process modeling, as only one user can edit a process model at a time. We propose a fragment-based locking mechanism for process models which supports increased collaboration while reducing the number of conflicts.

Users can lock individual fragments, upon which, any subsequent locking requests to those fragments will be denied. When a lock is requested for a fragment, we need to consider the lock granted for that fragment, as well as the locks of its ancestor and descendant fragments. To illustrate this, let us assume that a user requests a lock for F3 in Fig. 6 and that a lock has already been granted for its child fragment F6. If the requested lock is granted for F3, both F3 and F6 can be edited concurrently. As F3 contains F6, the user editing F3 can also edit the content of F6, which may result in a conflict with the edits done by the other user on F6. Thus, in this situation a lock for F3 cannot be granted. The same situation holds for the ancestor fragments of F3. If any ancestor fragment of F3 (e.g. F2) is locked, a lock for F3 cannot be granted. Thus, a fragment can only be locked if a lock has not yet been granted for that fragment and for any of its ancestor or descendant fragments. For example, two users can lock F3 and F7 at the same time. Concurrent updates to these two fragments do not cause conflicts, as neither of these fragments contain the other fragment. In this case, any subsequent lock request for fragments F3 and F7, and for their descendant and ancestor fragments will be denied.

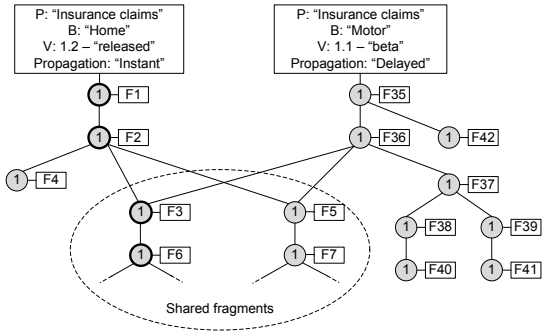

Fig. 6. Sharing fragments across different process model branches.

This fragment-based locking mechanism is realized by associating two locking attributes with each fragment: a boolean *direct lock* and an integer *indirect lock counter*.

A direct lock is assigned to a fragment that is directly locked by a user and gives the user the actual right to edit that fragment. The indirect lock counter is used to prevent conflicting lockings to descendant fragments. It is set to zero and incremented by one every time a descendant of the fragment in question is directly locked. A direct lock can only be placed if a fragment is not directly locked, its indirect lock counter is zero and none of its ancestor fragments is directly locked either. If so, the fragment is locked and the indirect lock counters of all its ancestors are incremented. Once a request for removing a lock is issued, the direct lock for that fragment is removed and the indirect lock counters of all its ancestor fragments are decremented. The indirect lock counter is required as multiple descendant fragments of a given fragment may be directly locked at the same time. In such situations, the counter of that fragment should not be reset until all direct locks of its descendant fragments have been released.

4 Controlled change propagation

In current process model repositories, similarity relations between different process models are not kept, so an update to a section of a process model remains confined to that process model, without affecting all process models of the repository that share (parts of) that section. This problem where two or more process models become "out-of-synch" is currently rectified manually, through maintenance cycles which are laborious and error-prone. For example, a team of business analysts at Suncorp was recently involved in a process consolidation effort between two of their insurance products, due to an update to one of the two products. However, it took them 130 man-hours to identify 25% of the shared fragments between the process models for these two products [23]. In fact, our experience tells us that real-life collections suffer from frequent mismatches among similar process models.

Since we reuse fragments across multiple process models, this provides a great opportunity to simplify the maintenance of the repository. For example, if a possible improvement is identified for fragment F3 of Fig. 6, that improvement can be made available immediately to both the Home and Motor insurance process models, since this fragment is shared by both these models. However, propagating fragment changes immediately to all affected process models may not be always desirable. Let us assume that the current version of the Motor insurance process model has been deployed in an active business environment. If an update to F3 has introduced an error, that error will immediately affect the Motor insurance process model, which could potentially impact important business operations. In order to prevent such situations, we support a flexible change propagation mechanism, where change propagations are controlled by a propagation policy associated with process model branches. The propagation policy of a process model branch can be set as either *instant propagation* or *delayed propagation*. If instant propagation is used in a branch, any change to any fragment in the current version of that branch is recursively propagated to all ascending fragments of that fragment in the current version, until the root fragment. Since the root fragment changes, a new version for that branch will be created, which will become the current version. If delayed propagation is used in a branch, changes to a fragment will not be immediately propagated throughout the current version. Instead, such changes will create pending updates for the current version. Then owners of the affected process model

are notified of all pending updates for that model. They can then review the pending updates and only trigger the necessary ones. Once a pending update is triggered, it will be propagated and a new version of the interested process model will be created.

Coming back to the example in Fig. 6, let us assume that the change propagation policy of the Home insurance branch is set to *instant* while that of the Motor insurance branch is set to *delayed*. If fragment F6 is updated (i.e. version 2 of F6 is created), new versions will instantly be created for all the ancestor fragments of F6 in the current version of Home (i.e. F3, F2 and F1, shown with a thicker border Fig. 6). As a new version is created for F1, which is the root fragment of Home, a new version of this process model will also be created, say version 1.3. On the other hand, since the Motor branch has a delayed propagation policy, new versions will not be created for the ancestor fragments of F6 in the current version of this branch. This means that F3 in Motor will still point to version 1 of F6, F36 to version 1 of F3 and F35 to version 1 of F36. Thus, the current version of Motor will still use version 1 of F6 and remain the same. However, the pending updates will be notified to the owner of the current version of Motor, who can decide whether or not to implement them.

Sometimes it is not required to create a new fragment version/process model version when a fragment is modified, e.g. after a fixing a minor error. Our technique supports such in-place editing of fragments, where the edited fragment version and all its ancestor fragments are updated without creating new versions. Changes performed in this mode will be available to all ancestor fragments instantly, irrespective of the change propagation policies.

5 Conceptualization of the storage structure

We now describe the conceptual model used to store our versioning system on top of a relational DBMS. The algorithms to populate and use this data structure, e.g. inserting or updating a fragment, are presented in Section 6.

An Object Role Modeling diagram of the storage structure is shown in Fig. 7. For illustration purposes, we populated this model with information from two process models: “Insurance claims” (the example used so far) and “Order processing”. Each process has two branches (e.g. Insurance claims has branches “Home” and “Motor”). Further, each branch has a root process model (i.e. the root Node), representing the first version of that branch. For example, the root process model of the Motor branch of the insurance claims process has node identifier N4 and refers to version number 1.0 having version name “alpha”. Each branch has a sequence of nodes where each node represents one version of a process model. Each node can have at most one immediate predecessor. For example, node N5 refers to version number 1.1 of its branch, and is the successor of node N4. The root node of a primary branch may optionally be derived from a node of an external process model branch (none in the sample population). The root node of a non-primary branch is always derived from a node of an internal process model branch. For example, the root node of the Motor branch (node identifier N4) is derived from node N2 of the Home branch.

Each node in a branch (i.e. each process model version) has an associated fragment version tree. In our example, the root fragment versions of process model versions 1.0 and 1.1 of the Home branch (i.e. nodes N1 and N2) are FV1 and FV6. FV1 and FV6

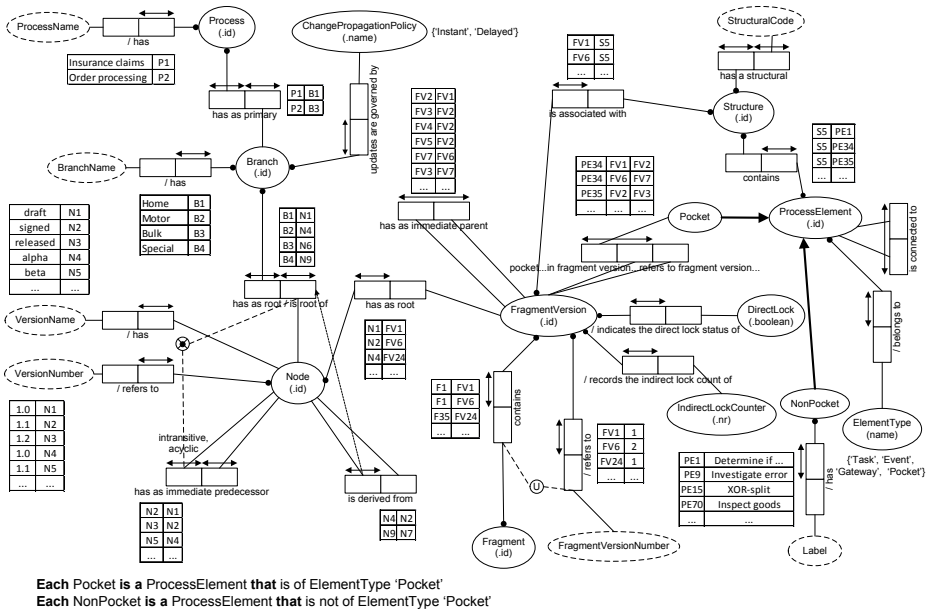

Fig. 7. Object-Role Modeling diagram of the storage structure.

are both contained in fragment F1 according to the sample population. Thus, FV1 and FV6 are two versions of the same fragment. In fact, FV1 is mapped to fragment version number 1 whilst FV6 is mapped to fragment version number 2 of F1. A fragment version can have multiple parents and children. For example, FV2 is the parent fragment of FV3, FV4 and FV5, while FV3 is the child of both FV2 and FV7. Hence, FV3 is shared between FV2 and FV7. A fragment version is associated with a structure which stores all process elements contained only in that fragment version. A structure is associated with a structural code, which is computed by considering its elements and their interconnections. The structural code is used to efficiently compare structures of fragments. Furthermore, two fragments can be efficiently compared by considering both structural codes and composition relationships. Process elements within structures can be of type non-pocket (i.e. tasks, events, gateways) and pocket. A pocked is a place holder for a child fragment. Continuing our running example, in fragment version FV1, pocket PE34 is mapped to fragment version FV2 while in FV6, PE34 is mapped to FV7. Thus, FV1 and FV6 share the structure S5 with different mapping for pocket PE34. Finally, the diagram models the association of change propagation policies with process branches and locking attributes with fragment versions.

As shown in the diagram of Fig. 7, we use a directed attributed graph of vertices (i.e. process elements) and edges (i.e. flow relations) to represent process models and fragments. Process elements can be tasks, events (e.g. timer or message events), gateways (e.g. AND-split, XOR-split, OR-join) and pockets. This meta-model is an extension of the canonical format used in the AProMoRe repository [24], where we introduced a new process element, namely the Pocket, to act as a placeholder for dynamically-computed child fragments. This abstract representation allows us to apply version control to pro-

cess models developed in multiple business process modeling languages (e.g. BPMN, YAWL, EPCs, BPEL), as well as to facilitate change propagation and concurrency control on those process models, regardless of their modeling language. For example, in order to version EPC models, we only have to convert EPCs to our representation format and vice versa. A full mapping between AProMoRe's canonical format and various process modeling languages is provided in [24]. We observe that in order to achieve language-independence, AProMoRe's canonical format covers only a set of concepts which are common to most process modeling languages.

6 Algorithms

In this section we describe the algorithms used in our repository for inserting, updating, retrieving and deleting process models and fragments.

6.1 Inserting and retrieving fragments

Process model insertion, retrieval and update methods of the repository depend on two main algorithms: AddFragment() (i.e. Algorithm 1) and FillFragment() (i.e. Algorithm 2). First, we will explain these two algorithms, which lay the foundation for the description of the other algorithms. AddFragment() method takes a process fragment as the input, stores the fragment and its child fragments in the repository and returns a unique identifier for the fragment. First, the AddFragment() method calls itself recursively to decompose the child fragments of the given fragment and to obtain identifiers for those child fragments. Then it replaces the subgraphs of all child fragments with pockets in order to obtain the structure of the given fragment. For each replaced child fragment, it adds a mapping from the added pocket identifier to its corresponding child fragment identifier. Thus, a fragment is represented in the repository as an structure and a set of *(pocket Id, child Id)* mappings.

Once the structure and the *(pocket Id, child Id)* mappings are obtained, we have to check whether there are similar components already stored in the repository, in order to prevent redundancies. First, the algorithm checks if a matching structure is already stored by invoking the GetMatchingStructureId() function. If a matching structure is found, we can check whether the same fragment is already stored, by searching for both the structure and the *(pocket Id, child Id)* mappings. This is achieved through application of the GetMatchingFragmentId() function. If a matching fragment identifier is found, the AddFragment() method returns the matched identifier without storing any information about the new fragment. If a matching structure is found and a matching fragment is not found, we can reuse the matched structure. Therefore, in that case, the algorithm adds a new fragment with the identifier of the matched structure and new *(pocket Id, child Id)* mappings. If a matching structure is also not found in the repository (which also implies that there are no matching fragments as well), the algorithm adds a new fragment with a new structure and new *(pocket Id, child Id)* mappings.

The FillFragment() method (listed in algorithm 2) is used to retrieve process fragments from the repository. It takes three parameters: fragment identifier, pocket identifier and a process model graph. This method retrieves the fragment identified by the given fragment identifier (i.e. first parameter) and fills the given process model graph

Algorithm 1: AddFragment

```
procedure AddFragment(Fragment f)
begin
    fragmentId ⇐ null
    childFragments ⇐ GetChildFragments(f)
    foreach childFragment in childFragments do
        pocketId ⇐ MakePocket(f, childFragment)
        childId ⇐ AddFragment(childFragment)
        pocketChildMappings ⇐ pocketChildMappings ∪ {(pocketId, childId)}
    matchingStructureId ⇐ GetMatchingStructureId(f)
    if matchingStructureId not null then
        matchingFragmentId ⇐
        GetMatchingFragmentId(matchingStructureId, pocketChildMappings)
        if matchingFragmentId not null then
            fragmentId ⇐ matchingFragmentId
        else
            fragmentId ⇐
            InsertFragment(matchingStructureId, pocketChildMappings)
    else
        structureId ⇐ AddStructure(f)
        fragmentId ⇐ InsertFragment(structureId, pocketChildMappings)
    return fragmentId
end
```

(i.e. third parameter) with the content of the fragment. The purpose of the second parameter (i.e. pocket identifier) will be explained later. It is possible to invoke the FillFragment() method either by providing a valid process model graph or by providing an empty process model graph (i.e. null). If an empty process model graph is provided, the fragment will be constructed as a new process model graph. If a process model graph is provided, the fragment will be composed into a pocket of the given process model graph. The pocket to be used for the composition is identified by the pocket identifier given as the second parameter of the FillFragment() method. When the FillFragment() method is invoked to compose a fragment, it is invoked by providing an empty process model graph, which forces it to construct the fragment as a new process model graph. Once the FillFragment() method is invoked, it retrieves the structure of the requested fragment and assigns it to the given empty process model graph. Then the FillFragment() method retrieves the *(pocket Id, child Id)* mappings of the requested fragment from the repository. Now all the pockets in the structure of the requested fragment have to be filled with process model graphs of its child fragments. For this, the FillFragment() method invokes itself recursively for each (pocket Id, child Id) mapping. In each such invocation, the process model graph of the requested fragment and the identifier of the pocket to be replaced with the child fragment are provided as an input, in addition to the identifier of the child fragment. This forces the method to replace pockets of the graph with child fragments. Such recursive invocations replaces all pockets in the graph with descendant fragments, thus completing the process model graph of the requested fragment.

Algorithm 2: FillFragment

```
procedure FillFragment(fragmentId, pocketId, ProcessModel p)
begin
    ProcessModel f ⇐ GetStructure(fragmentId)
    if p not null then
        ReplacePocket(p, pocketId, f)
    else
        p ⇐ f
    pocketChildMappings ⇐ GetPocketChildMappings(fragmentId)
    foreach (childPocketId, childId) in pocketChildMappings do
        FillFragment(childId, childPocketId, p)
end
```

6.2 Inserting a new process model

Next, we will explain the algorithms for manipulating process models and fragments with reference to the FillFragment() and AddFragment() methods. The AddProcessModel() method adds a new process model to the repository, which takes the process model graph and the name of a new process model. As this method adds a new process model, first it creates the main branch of the process model using the CreateMainBranch() method. Then the RPST fragment tree of the given process model is created using the ComputeRPST() method. The ComputeRPST() method returns the root fragment of the computed fragment tree. Once the root fragment is available, the AddProcessModel() method invokes the AddFragment() method to store the root fragment and it descendant fragments in the repository. The AddFragment() method returns an identifier for the root fragment as mentioned in section 6.1. Once the identifier of the root fragment is available, the AddProcessModel() method adds a tuple *(process model name, branch name, root fragment identifier)* to the repository to represent the new process model using the InsertProcessModel() method.

Algorithm 3: Add Process Model

```
procedure AddProcessModel(ProcessModel p, processModelName)
begin
    branchName ⇐ CreateMainBranch(processModelName)
    rootFragment ⇐ ComputeRPST(p)
    rootFragmentId ⇐ AddFragment(rootFragment)
    InsertProcessModel(processModelName, branchName, rootFragmentId)
end
```

6.3 Checking out process models and fragments

Now we will go through the algorithms for checking out process models and fragments from the repository. In fact, there is no difference between checking out a fragment and

checking out a process model. As any process model is stored as a fragment tree, checking out a process model is equivalent to checking out its root fragment. Therefore, we will only discuss the CheckoutFragment() method, which is used to retrieve the fragment identified by a given fragment identifier. The CheckoutFragment() method first initializes an empty process model graph, which is used to hold the process model graph of the requested fragment. Before checking out a fragment, the requested fragment and its ancestor fragments have to be locked in order to prevent possible conflicts (see section 3). First, we have to check whether the requested fragment is directly locked, as we can't place a direct lock if the fragment is already locked directly. This check is performed by invoking the IsDirectLocked() method, which returns true if the given fragment is directly locked, and returns false if it is not directly locked. Then the IncrementAncestorIndirectLocks() method is invoked to recursively increment indirect locks of all ancestor fragments of the given fragment. This method (listed in Algorithm 5) returns true if the indirect locks could be incremented in all ancestors, and returns false if an indirect lock of any ancestor fragment could not be incremented. If both IsDirectLocked() and IncrementAncestorIndirectLocks() methods return true for the given fragment, the CheckoutFragment() method places a direct lock on the requested fragment (by calling the PlaceDirectLock() method) and invokes the FillFragment() method to recursively construct the process model graph of the requested fragment as mentioned in section 6.1.

Algorithm 4: Checkout Fragment

```
procedure CheckoutFragment(fragmentId)
begin
  ProcessModel p ⇐ null
  if not IsDirectLocked(fragmentId) and
  IncrementAncestorIndirectLocks(fragmentId) then
    PlaceDirectLock(fragmentId)
    FillFragment(rootFragmentId, null, p)
  return p
end
```

6.4 Updating process models and fragments

We can now discuss the algorithms for updating (i.e. checking in) process models and fragments. As we did in the previous section, describing only the algorithm for checking in a fragment is sufficient, as checking in a process model is equivalent to checking in its root fragment. First, we have to study the procedure for updating a fragment. A user checks out a fragment (using the CheckoutFragment() method) for updating by providing its fragment identifier. The repository returns the process model graph of the requested fragment. Then the user can update the process model graph as necessary. Once the required modifications to the process model graph are completed, the user can check in the fragment using the CheckinFragment() method, discussed in this section.

Algorithm 5: Increment Ancestor Indirect Locks

```
procedure IncrementAncestorIndirectLocks(fragmentId)
begin
    parentIds ⇐ GetCurrentParentFragmentIds(fragmentId)
    foreach parentId in parentIds do
        if not IsDirectLocked(parentId) and
        IncrementAncestorIndirectLocks(parentId) then
            IncrementIndirectLock(parentId)
            return true
        else
            return false
    return true
end
```

The CheckinFragment() method takes the fragments identifier used to check out the fragment and the modified process model graph as inputs. First the CheckinFragment() method decomposes the modified process model graph using the AddFragment() method, which stores all fragments of the modified graph and returns an identifier for the modified fragment. If the updated fragment is used as the root fragment of process models, new versions have to be created for those process models with the updated root fragment. The CheckinFragment() method does this by invoking the CreateNewVersion() method for each affected process model. Then the direct lock placed on the updated fragment is cleared using the RemoveLock() method.

Now the CheckinFragment() method should create new fragments for ancestor fragments of the updated fragment. However, we only have to consider the ancestor fragments in which the indirect lock count is greater than zero. There could be ancestor fragments with zero indirect locks, as ancestor fragments will not be indirectly locked if they do not belong to a current version of a process model. In order to perform this propagation, the CheckinFragment() method calls the PropagateToParents() method for each indirectly locked parent fragment, which in turn recursively invokes itself to propagate changes to all ancestor fragments and decrement their indirect locks.

The PropagateToParents() method (listed in Algorithm 7) takes three parameters: identifier of the parent to which the changes have to be propagated, identifier of the old version of the changed child fragment and the identifier of the new version of the changed child fragment. We have to create a new version of the given parent fragment, as one of its children was changed. However, all other details of the parent fragment remains the same between its old and new version except the updated child relationship. Therefore, the new version of the parent fragment is created by copying its version using the CopyFragment() method. This method copies all details of the old parent fragment, including its structure, child relationships, direct lock status and indirect lock count. Then the PropagateToParents() method decrements the indirect lock count of the new parent fragment (which has the indirect lock count of the old parent fragment) by calling the DecrementIndirectLock() method. After that, the child relationships of the new parent fragment are updated by replacing the old child fragment identifier (given as the second input parameter) with the new child fragment identifier (given as the third

input parameter). As the given parent fragment can be the root fragment of some process models, the algorithm retrieves all affected process models and creates new versions for all those process models using the CreateNewVersion() method. Once a new parent fragment is created, all child fragments of the old parent fragment consider the new parent fragment as the current version. Therefore, the indirect lock count associated with the old parent fragment is cleared using the ClearIndirectLock() method, as it is no longer used by existing transactions. Once these steps are performed, propagation of changes of one ancestor level is completed. Then the algorithm retrieves all indirectly locked parent fragments of the given parent fragment and calls itself for all those parent fragments in order to propagate changes recursively until root fragments are reached.

Algorithm 6: Checkin Fragment

```
procedure CheckinFragment(oldFragmentId, ProcessModel p)
begin
    newFragmentId ⇐ AddFragment(p)
    processModelIds ⇐ GetUsedProcessModelIds(oldFragmentId)
    foreach processModelId in processModelIds do
        CreateNewVersion(processModelId, newFragmentId)
    RemoveLock(oldFragmentId)
    lockedParentIds ⇐ GetIndirectlyLockedParentIds(oldFragmentId)
    foreach parentId in lockedParentIds do
        PropagateToParents(parentId, oldFragmentId, newFragmentId)
end
```

Algorithm 7: Propagate To Parents

```
procedure PropagateToParents(parentId, oldFragmentId, newFragmentId)
begin
    newParentId ⇐ CopyFragment(parentId)
    DecrementIndirectLock(newParentId)
    ReplaceChild(newParentId, oldFragmentId, newFragmentId)
    processModelIds ⇐ GetUsedProcessModelIds(oldFragmentId)
    foreach processModelId in processModelIds do
        CreateNewVersion(processModelId, newFragmentId)
    ClearIndirectLock(oldFragmentId)
    lockedParentIds ⇐ GetIndirectlyLockedParentIds(oldFragmentId)
    foreach parentId in lockedParentIds do
        PropagateToParents(parentId, oldFragmentId, newFragmentId)
end
```

6.5 Deleting process models and fragments

Similar to the other operations, deletion is also similar for process models and fragments (i.e. deleting a process model is equivalent to deleting the root fragment of the process model). The DeleteFragment() method takes the identifier of a fragment as the input. First, it checks whether the given fragment is used as the root fragment of any process model or as a child fragment of any other fragment. If the given fragment is used in any of the above, it will not be deleted and the algorithm returns false to indicate that the deletion is not successful. If it does not have any usages, its *(pocket Id, child Id)* mappings are deleted. However, the structure of the fragment is deleted, only if it is not shared by any other fragment. After deleting the *(pocket Id, child Id)* mappings and the structure, the DeleteFragment() method calls itself recursively for each child fragment to delete all unshared child fragments of the given fragment. Return values of these recursive invocations of the DeleteFragment() method are ignored as deletion of some child fragments may not be successful if child fragments have multiple parents. Therefore, deletion of child fragments continues even if some child fragments could not be deleted. Once all possible child fragments of the given fragment are deleted, the DeleteFragment() method returns true to indicate the success of the deletion.

Algorithm 8: Delete Fragment Version

```
procedure DeleteFragment(fragmentId)
begin
  if GetNumberOfUsedProcessModels(fragmentId) = 0 and
  GetNumberOfParents(fragmentId) = 0 then
    structureId ⇐ GetStructure(fragmentId)
    childIds ⇐ GetChildIds(fragmentId)
    RemovePocketChildMappings(fragmentId)
    if GetNumberOfUsedFragments(structureId) = 0 then
      RemoveStructure(structureId)
    foreach childId in childIds do
      DeleteFragment(childId)
    return true
  else
    return false
end
```

7 Evaluation

We implemented the proposed versioning model and associated storage structure in Java on top of the MySQL DBMS, and used this prototype to evaluate our technique. We conducted the experiments on two industrial process model collections: 595 EPC models from the SAP R/3 reference model and 248 EPC models from IBM's BIT library.[5]

[5] http://www.zurich.ibm.com/csc/bit/downloads.html

First, we measured the gain induced by vertical sharing. We took a set of models with varying size (ranging from 25 to 107 nodes for the SAP dataset and from 10 to 40 nodes for the IBM dataset), and for each of them we created 100 subsequent versions by randomly updating a set of adjacent nodes (i.e. localized changes). We allowed four types of basic change operations with corresponding probabilities: change task label (33%), delete task (33%), insert a task between two adjacent nodes (17%) and insert a task in parallel to another task (17%). These probabilities were chosen to balance insertions and deletions so as to prevent excessive growth or shrinkage of a process model, thus simulating localized changes. For each model, we repeated the experiment by changing 5%, 20% and 50% of the models' size. After creating a new version, we calculated the vertical storage gain G_v compared to storing full process model versions. Let N be the number of nodes for storing full versions and N_v the number of nodes stored if sharing fragments vertically. Then $G_v = (N - N_v) \cdot 100/N$. Fig. 8 reports the average G_v for each dataset, by aggregating the values of all changed process models. Our technique incurs a slight initial overhead due to storing pockets and edges connecting pockets. However, the vertical storage gain rapidly increases as we add new versions. For the SAP dataset it levels off at 82% for small updates (5% of model size), and 55% for larger updates (50% of size) whilst for the IBM dataset it levels off at 78% for small updates and 46% for larger updates. This confirms our intuition that storing duplicate fragments only once across different process model versions can dramatically reduce the overall repository size.

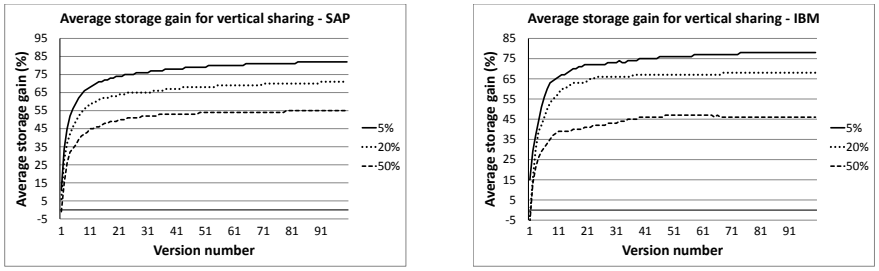

Fig. 8. Average storage gain when sharing fragments across versions of the same process model.

Second, we measured the gain G_h induced by horizontal sharing. For each dataset, we randomly inserted all process models in the repository, and as we increased the size of the repository, we compared the size of storing duplicate fragments only once with the size of storing full process models. We only counted the size of maximal fragments across different process models, i.e. we excluded child fragments within shared fragments. Let N be the number of nodes for storing full process models, F the set of fragments, N_f the number of nodes of fragment f and O_f the number of its occurrences. Then $G_h = \sum_{f \in F} N_f \cdot (O_f - 1)/N \cdot 100$. Fig. 9a shows the results of this experiment. As expected, the horizontal gain increases with the number of process models reaching a final value of 35.6% for the SAP dataset and 21% for the IBM dataset. This trend is determined by the increasing number of shared fragments as the total size of the repository increases. For example, for the SAP dataset there are 98 shared fragments

when the repository is populated with 100 process models and this number increases to 840 fragments with the full dataset. This gives an indication of the reduction in maintenance effort, as any update to any of those fragments or their child fragments, will be automatically reflected onto all process models containing those fragments.

Following from the results of the previous experiment, we tested the effects of change propagation onto the repository. We populated the repository with the SAP dataset and performed 100 updates on randomly selected fragments. An update to a fragment consists of a combination of the following operations with associated probabilities: label change (33%), serial node insertion (17%), parallel node insertion (33%) and node deletion (33%). The total number of operations performed in an update is proportional to the number of nodes in the fragment being updated. In these tests we set the operations-to-nodes ratio to one. For example, when updating a fragment with 10 nodes, 10 operations were performed consisting of approximately 3 label changes, 3 node deletions, 2 serial node insertions and 2 parallel node deletions.

The change propagation policy of all process models was set to instant propagation during these tests as we wanted all changes to be immediately propagated to all affected models. After each update, we measured the total number of automatically propagated changes in the repository. We repeated the same experiment for the IBM dataset. The average results for 10 test runs with both datasets are shown in Fig. 9b. Accordingly, the number of propagated changes increases with the number of updates performed on a process model collection. For example, on average 20 automatic changes were applied by the repository across different process models when 100 updates were performed on the SAP dataset. If our change propagation method is not used, process modelers have to analyze the entire process model collection and apply all these changes to relevant process models manually, which could be a time consuming and error-prone activity. Thus, automatic change propagation provides indeed a significant benefit in maintaining the consistency of the repository.

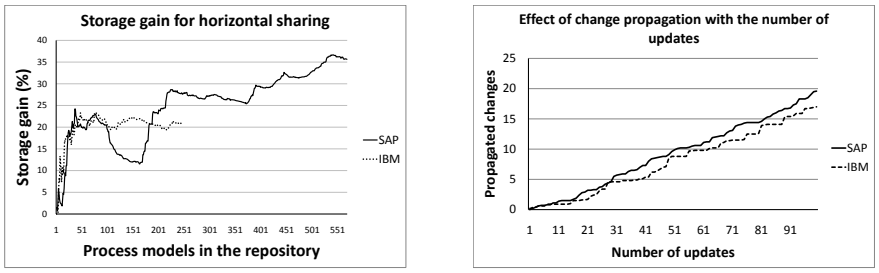

Fig. 9. Vertical storage gain (a) and change propagation (b) with the growth of the repository.

Finally, we measured the effectiveness of our fragment-based locking by comparing it with the model-based locking available in current process model repositories. In this experiment, we used software agents to randomly lock fragments of a given process model collection in order to simulate random updates. We first generated a sequence of locking actions for each agent and saved it in a file. An action is a tuple *(process model identifier, fragment identifier, locking duration)*. For example action (12, 25, 560)

forces an agent to lock fragment 25 of process model 12 for 560 milliseconds. For each action, the process model was selected using a uniform probabilistic distribution over all process models in a given collection. The fragment was selected based on a Gaussian distribution over the sizes of the fragments of the selected process model, where the mean size of the distribution was set to 10% of the size of the selected process model. The locking duration was determined based on an inverse exponential distribution with mean of 5 seconds, in order to speed up the tests.

Once all action files were generated, we executed two tests for each file: i) each agent attempted to lock only the specified fragment; ii) each agent attempted to lock the entire process model for each action, to simulate the traditional model-based locking. We executed these tests for two process model collections, with 10 and 30 process models, chosen with uniform size distribution from the SAP dataset. We used these small numbers of process models as in an average BPM project multiple users typically work collaboratively on a small set of process models. For each collection, we performed three tests by varying the number of concurrent agents from 10, to 20 and 30, and we computed the success rate for each test as the ratio of the number of successful operations over the number of total operations. The results are shown in Fig. 10.

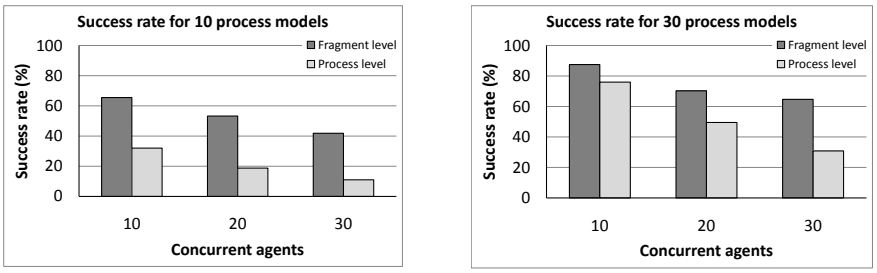

Fig. 10. Success rate of locking operations in 10 process models (a) and 30 process models (b).

As expected, the fragment-based locking mechanism scored the highest success rate in all tests. We also observed that the gain of this locking compared to that of model-based locking increases with the increase of concurrent agents (for example, when using 10 agents on 30 process models, fragment level locking facilitated 15% more operations than process level locking, while fragment level locking facilitated 110% more operations for 30 agents). Further, this gain is higher when agents are competing for a smaller number of process models. Thus, we can conclude that our fragment-based locking mechanism is more effective than the traditional model-based locking.

8 Related work

In this section we discuss related work in the field of BPM as well as in other fields, such as software engineering and computer aided design. Our discussion is categorized under version control, repositories, process model changes and concurrency control.

8.1 Version control

Version control has been extensively studied in at least three different fields: Temporal Databases (TDBs), Software Engineering (SE) and Computer Aided Design (CAD). TDBs [34, 13] deal with issues that arise when data evolution and histories of temporal models have to be managed. In SE, Source Code Control System (SCCS) [33] was probably one of the precursors of version control systems. Here a *revision* of a file is created each time the file is modified. Revision Control Systems (RCS) [37] extended SCCS by introducing the concept of *variant* to capture branching evolution (e.g. in SCCS, evolutions are represented as a sequence, while in RCS they are represented as a tree). Space consumption is optimized by only storing textual differences (*deltas*) between subsequent versions. This is the same approach used by popular version control systems such as CVS and SVN. It is possible to use textual deltas to version control process models by considering XML based serializations of process models (e.g. EPML, XPDL, YAWL). However, such deltas only serve as a method to reconstruct different versions and do not facilitate other essential aspects of process model repositories as mentioned later in this section.

Within SE, approaches in the area of Software Configuration Management [8], propose to use database technology to enhance the underlying data model and make the notion of version explicit. *Damokles* [12] is probably one of the first database-based versioning environment for SE. It offers the notion of *revision* as a built-in datatype and a version-aware data modeling language. In [29] the authors present an object graph versioning system (HistOOry) which allows applications to store and efficiently browse previous states of objects. This approach keeps history of object graphs, while ours deals with version control of graphs. Moreover, our goals are different: we focus on graph fragment reusability and update propagation.

A version control method specifically designed for process models is proposed in [1]. This method is based on change operations: the differences between two process model versions are specified as a set of insert, delete and modify operations on tasks, links and attributes. The version history of a process model is stored as the initial version plus the set of change operations required to derive all subsequent versions. When a new process model version is checked in, the change operations required to derive this version from the last version of the same process model are computed and stored as the delta of the new version. Similarly, when a process model version is checked out, all change operations required to derive the requested version from the initial version are retrieved and applied to the initial version to construct the requested version. Another method for process model version control is to store all versions of a process model in a single graph by annotating the graph's nodes and edges with version numbers [43]. Once such a graph is built, one can derive any version of its process model by following a set of derivation rules. Thus, deltas between process model versions are captured as a set of graph elements (i.e. nodes and edges). However, the types of deltas proposed in the above two methods, as well as the textual deltas used in SCCS, RCS, CVS and SVN discussed earlier, do not have any other purpose than reconstructing different versions. In contrast, we use process fragments as deltas, which are meaningful components of process models. In addition to reconstructing different versions, we use fragments to automatically propagate changes across process model versions and across different process models, and to reduce conflicting edit operations over these models.

Further, fragments can be used as queries for searching specific process models in large repositories, as done in [38], or as compositional units to create new process models. For example, a fragment used in an old process model version can be reused in a new version of another process model. Hence, we argue that our fragment-based approach is better-suited for the management of process models, specially when other requirements such as change propagation, concurrency control and search are considered, in addition to pure version control.

Thomas [36] presents an architecture for managing different versions of reference process models. However this approach focuses on high-level aspects of versioning such as integration with different enterprise databases, inter-connections with external applications, attributes to be associated with versions and user interface design. Thus, this work is complementary to our research as our methods can be embedded in such an architecture.

8.2 Repositories

Repositories provide a shared database for artifacts produced or used by an enterprise, and also facilitate functions such as version control, check-in, check-out and configuration management [5]. The use of repositories for managing artifacts in different domains has been studied and different storage mechanisms have been proposed. The concept of managing complex artifacts as aggregations of lower level components has been discussed in the literature (e.g. [8, 16, 18, 17]). In particular, version control and change propagation of such composite artifacts have been studied in the context of CAD repositories [16, 18, 17]. Accordingly, the highest degree of sharing is obtained when all software components are versioned including composite and atomic components, and their relationships. The storage technique that we propose extends such concepts in the context of process model management. Most of the research on composite artifact storage mechanisms assumes that lower level objects and their composition relationships are explicitly stated by users. In our technique, we use the RPST algorithm to automatically decompose process models into lower level fragments in linear time. Further, when storing process models we always decompose them into the smallest possible RPST fragments, thus increasing the advantages of space utilization, change propagation and concurrency control. We also share the structures and composition relations between such process models. This allows us to maximize the sharing of fragments among process models (i.e. identical structures are shared even if child mappings are not the same). Further, we share components (i.e. fragments) and structures across multiple versions (i.e. vertically) as well as across different process models (i.e. horizontally).

Business process model repositories stemming from research initiatives support process model-specific features in addition to basic insert, retrieve, update and delete functions [26, 35, 27, 7, 42], such as searching stored process models based on different parameters. For example, the semantic business process repository [27] focuses on querying business processes based on ontologies while the process repository proposed in [7] also focuses on the lifecycle management of process models. Similar features can be found in commercial process model repositories, such as the ARIS platform [9]. However, both academic and commercial process model repositories only support basic version control at the level of process nodes. Moreover, none of these solutions

adequately addresses the problems of change management and concurrency control. For example, in ARIS one can only propagate updates to node attributes.

Redundant process fragments are identified as an issue when managing large process model repositories in [40]. If these fragments are not kept in synch, changes to the repository may lead to inconsistencies. Since we share redundant fragments only once, and we propagate changes across them, our technique can be seen as a way of solving the "redundant process fragments" issue described in [40].

8.3 Process model changes

Different classifications of process model changes have been proposed in the literature [41, 10, 11]. Weber et al. [41] propose a set of change patterns that can be applied to process models and process instances, in order to align these artifacts with changing requirements. These change patterns focus on fragment-level operations (e.g. inserting a new fragment into a process model, deleting a fragment or moving a fragment to a different position) as well as on control-flow changes (e.g. adding a new control-flow dependency and changing the condition of a conditional branch). The classification proposed by Dijkman [10, 11] focuses on finer-grained changes including the insertion and removal of an activity, the refinement of an activity into a collection of activities and the modification of an activity's input requirements. This classification also includes changes performed on resource-related aspects, such as allocating an activity to a different human role. These classifications are useful for many areas, such as developing and evaluating process model editors, identifying differences between process models, designing change operation based concurrency control techniques and developing version control systems. However, our storage and version control technique considers the final states of process models, and the operations applied to derive different process models are not required for our approach. As such, this work is complementary to ours. In fact, we do not impose any restriction on the type of changes that can be performed on our process models.

8.4 Concurrency control

Fine-grained locking of generic objects and CAD objects has been studied in [28, 2, 3]. However, the possibility of fine-grained locking of process models at the process fragment level has not been studied in the literature. The issue of resolving conflicts in different process model versions has been explored both at design-time and at run-time. At run-time, the propagation of process model changes to running process instances without causing errors and inconsistencies has been extensively studied in the literature [32, 31, 15, 20]. Since our process models are design-time artifacts, this work is complimentary to ours. At design-time, Küster et al. [22, 21, 14] propose a method for merging two versions of the same process model based on the application of *change operations* which can be automatically identified without the need for a change log. Similar to our approach, this solution relies on the decomposition of process models into SESE fragments. However, this approach focuses on resolving conflicts once overlapping modifications are detected, while our approach prevents conflicts before they occur through selective locking. Thus, it may be possible to combine both approaches in order to develop flexible collaborative environments.

9 Conclusion

This paper presents a novel versioning model and associated storage structure specifically designed to deal with (large) process model repositories. The focal idea is to store and version single SESE process fragments, rather than entire process models. The motivation comes from the observation that process model collections used in practice feature a great deal of redundancy in terms of shared process fragments.

The contribution of this technique is threefold. First, repository users can effectively keep track of the relations among different process models (horizontal sharing) and process model versions (vertical sharing). Second, sophisticated change propagation is achieved, since changes in a single fragment can be propagated to all process models and process model versions that share that fragment. This goes well beyond the change propagation provided by current process model repositories. This in turn allows users to automatically ensure consistency, and maximize standardization, in large process model repositories. Finally, locking can also be defined at the granularity of single fragments, thus fostering concurrent updates by multiple users, since it is no longer required to lock entire process models. To the best of our knowledge, fragment-based concepts have not been adopted to study these aspects of process model collections to date.

An important application of our technique is the management of variability in process model repositories. In fact, variants of a same process model, e.g. the "Home" and "Motor" variants of an "Insurance claim" process model, are never that dissimilar from each other, i.e. they typically share various fragments [23]. These variants can either be explicitly modeled as different branches of the same process, or their commonalities can be automatically detected when these variants are inserted into the repository. In both cases, our technique will trace these links among the variants, and keep the variants synchronized whenever they undertake changes.

This technique was implemented and its usefulness was evaluated on two industrial process model collections. In future work, we plan to combine our fragment-based locking method with operational merging [25] to provide more flexible conflict resolution in concurrent fragment updates. We also plan to version process' data and resources, and to further evaluate our technique by conducting usability tests with process model repository users.

Acknowledgments This research is funded by the Smart Services Cooperative Research Centre (CRC) through the Australian Government's CRC Programme.

References

1. H. Bae, E. Cho, and J. Bae. A version management of business process models in bpms. In *APWeb/WAIM Workshops*, pages 534–539, 2007.
2. F. Bancilhon, W. Kim, and H. F. Korth. A model of cad transactions. In A. Pirotte and Y. Vassiliou, editors, *VLDB*, pages 25–33. Morgan Kaufmann, 1985.
3. N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3):269–317, 1991.
4. B. Berliner and I. Prisma. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990.
5. P. A. Bernstein and U. Dayal. An overview of repository technology. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB*, pages 705–713. Morgan Kaufmann, 1994.

6. J. Cardoso. Poseidon: a Framework to Assist Web Process Design Based on Business Cases. *Int. J. Cooperative Inf. Syst.*, 15(1):23–56, 2006.
7. I. Choi, K. Kim, and M. Jang. An xml-based process repository and process query language for integrated process management. *Knowledge and Process Management*, 14:303–316, 2007.
8. R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.
9. R. Davis and E. Brabänder. *ARIS design platform: getting started with BPM*. Springer-Verlag New York Inc, 2007.
10. R.M. Dijkman. A classification of differences between similar BusinessProcesses. In *edoc*, page 37. IEEE Computer Society, 2007.
11. R.M. Dijkman. Diagnosing differences between business process models. In *BPM*, pages 261–277, 2008.
12. K.-R. Dittrich. The damokles database system for design applications: its past, its present, and its future. pages 151–171. Ellis Horwood Books, 1989.
13. M. Dumas, M.-C. Fauvet, and P.-C. Scholl. TEMPOS: a platform for developing temporal applications on top of object DBMS. *IEEE TKDE*, 16(3), 2004.
14. C. Gerth, J.-M. Kster, M. Luckey, and G. Engels. Precise detection of conflicting change operations using process model terms. In *Proc. of MODELS*, volume 6395 of *LNCS*, 2010.
15. G. Joeris and O. Herzoz. Managing evolving workflow specifications. In *Proc. of IFCIS*, pages 310–321. IEEE, 1998.
16. R.-H. Katz. Towards a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–408, 1990.
17. R. H. Katz and E. E. Chang. Managing change in a computer-aided design database. In *VLDB*, pages 455–462. 1987.
18. R. H. Katz, E. E. Chang, and R. Bhateja. Version modeling concepts for computer-aided design databases. In C. Zaniolo, editor, *SIGMOD Conference*, pages 379–386. ACM Press, 1986.
19. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.
20. D. Kim, M. Kim, and H. Kim. Dynamic business process management based on process change patterns. In *Proc. of ICCIT*, pages 1154–1161. IEEE, 2007.
21. J.-M. Küster, C. Gerth, and G. Engels. Dependent and conflicting change operations of process models. In *Proc. of ECMDA-FA*, pages 158–173, 2009.
22. J.-M. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and resolving process model differences in the absence of a change log. In *Proc. of BPM*, volume 5240 of *LNCS*, pages 244–260, 2008.
23. M. La Rosa, M. Dumas, R. Uba, and R.M. Dijkman. Merging business process models. In *Proc. of OTM*, volume 6426 of *LNCS*, pages 96–113, 2010.
24. M. La Rosa, H.A. Reijers, W.M.P. van der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, and L. Garcia-Banuelos. Apromore: An advanced process model repository. *ESWA*, 2011.
25. E. Lippe and N. van Oosterom. Operation-based merging. *SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.
26. C. Liu, X. Lin, X. Zhou, and M. E. Orlowska. Building a repository for workflow systems. In *TOOLS (31)*, pages 348–357. IEEE Computer Society, 1999.
27. Z. Ma, B. Wetzstein, D. Anicic, S. Heymans, and F. Leymann. Semantic business process repository. In M. Hepp, K. Hinkelmann, D. Karagiannis, R. Klein, and N. Stojanovic, editors, *SBPM*, volume 251 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
28. J. P. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *CSCW*, pages 278–287, 1996.
29. F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: efficient objet graph versioning. In *OOPSLA 2009*, Orlando, Florida, USA, 2009.

30. H. A. Reijers, S. van Wijk, B. Mutschler, and M. Leurs. BPM in Practice: Who Is Doing What? In *Proc. of BPM*, pages 45–60. Springer, 2010.
31. S. Rinderle, M. Reichert, and P. Dadam. Disjoint and overlapping process changes: Challenges, solutions, applications. In R. Meersman and Z. Tari, editors, *CoopIS/DOA/ODBASE (1)*, volume 3290 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2004.
32. S. Rinderle, M. Reichert, and P. Dadam. Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases*, 16(1):91–116, 2004.
33. M.-J. Rochkind. The source code control system. *IEEE TSE*, 1(4):364–370, 1975.
34. R. T. Snodgrass. Temporal databases. In *Proc. of GIS*, 1992.
35. M. Song, J. A. Miller, and I. B. Arpinar. Repox: An xml repository for workflow designs and specifications. Technical report, Univeristy of Georgia, USA, 2001.
36. O. Thomas. Design and implementation of a version management system for reference modeling. *JSW*, 3(1):49–62, 2008.
37. W.-F. Tichy. Design implementation and evaluation of a revision control system. In *Proc. of the 6th Int. Conf. on Software Engineering*, Tokyo, Japan, 1982.
38. R. Uba, M. Dumas, L. Garcia-Banuelos, and M. La Rosa. Clone detection in repositories of business process models. In *BPM*. Springer, 2011.
39. Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.
40. B. Weber, M. Reichert, J. Mendling, and H. A. Reijers. Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486, 2011.
41. B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.*, 66(3):438–466, 2008.
42. Z. Yan, R. M. Dijkman, and P. W. P. J. Grefen. Business process model repositories - framework and survey. Technical report, Eindhoven University of Technology, The Netherlands,, 2009.
43. X. Zhao and C. Liu. Version management in the business process change context. In *Proc. of BPM*, volume 4714 of *LNCS*, pages 198–213, 2007.