GROWING TREES – A VERSIONING APPROACH FOR BUSINESS PROCESS MODELS BASED ON GRAPH THEORY

Clever, Nico, University of Muenster – ERCIS, Leonardo-Campus 3, 48149 Münster, DE, nico.clever@ercis.uni-muenster.de
Holler, Justus, University of Muenster – ERCIS, Leonardo-Campus 3, 48149 Münster, DE, justus.holler@ercis.uni-muenster.de
Püster, Johannes, University of Muenster – ERCIS, Leonardo-Campus 3, 48149 Münster, DE, johannes.puest@ercis.uni-muenster.de
Shitkova, Maria, University of Muenster – ERCIS, Leonardo-Campus 3, 48149 Münster, DE, maria.shitkova@ercis.uni-muenster.de

Abstract

This paper examines parallels in version management between software engineering and process modelling. Best practices in software engineering version management, identified by a literature review, are discussed in the context of process modelling. Based on the results, a concept for version management in process modelling is derived. We present both a versioning approach for business process models as well as an XML storage format to serve as a foundation for an implementation of our approach. The versioning concept at hand is based on graph theory and strives to serve as a framework for future model versioning research and implementation in process modelling tools.

Keywords: process modelling, versioning, software engineering, graph theory, XML
1 Introduction

The topics and approaches in the area of software engineering and process modelling within the information systems discipline resemble themselves to a significant extent. In both areas, fragmentary depictions of the reality—models—are created in order to serve a specific purpose. In both disciplines models represent an excerpt of the reality to abstract from unnecessary details and to highlight relevant information. In software engineering they serve to support the development of software. In the following, we will focus on process models, which are used within many if not all areas of business process management.

Version management has been of high interest within the software engineering area since decades (Rochkind, 1975; Tichy, 1982; Conradi and Westfechtel, 1998; Pluquet et al., 2009). Software engineering practitioners and researchers have created a multitude of approaches and tools, which support versioning over the course of a software development project. Critical features include redo and undo functionality, merging of different implementation branches and version histories. However, unlike in software engineering, version management in process modelling has only recently become a topic in the information systems community (Zhao and Liu, 2007; Küster et al., 2010; Thomas, 2008; Ekanayake et al., 2011). Contributions to version management of conceptual models are therefore only marginal, although there is support for versioning in some tools, e.g. (Signavio GmbH, 2012).

When it comes to collaborative business process modelling, well defined and conflict-free versioning mechanisms are inevitable to prevent hazardous modelling and lost information (Hadar and Soffer, 2006; Batini et al., 1986). We therefore argue that distributed collaborative business process modelling projects will heavily benefit from a suitable implementation of versioning techniques for process models. To bridge this research gap, we propose an approach for efficient and conflict-free management of changes in process models. As there are already established approaches and solutions for version management within the area of software engineering, we pose the following research question:

*How can the existing concepts of version management in software engineering be transferred to process model version management?*

The remainder of this paper is structured as follows. We describe our research methodology in section two and subsequently review the literature in section three. Based on this review we propose a concept for process model versioning and discuss implementation requirements in section four. The paper concludes with summary, outlook and future research in section five.
2 Research Methodology

The proposed approach as presented in this article follows the design science research methodology paradigm introduced by (Peffers et al., 2007) that requires six consecutive phases. As the last phase of communicating the research results is covered in this paper, it is omitted in Figure 1, which illustrates how we address each phase. To identify the problem and motivate the research, we conducted a literature review in the areas of software engineering and process modelling and discuss different approaches to versioning. Subsequently, we argue for transferring concepts from software engineering to process modelling to define our objective. We then present our concepts for serialization, versioning and discuss further implementation requirements. However, actual demonstration through implementation and evaluation will be left for future research.

Figure 1. Research Methodology based on (Peffers et al., 2007)

3 Literature Review

Version (or revision) control systems are widely used in software engineering practice (Ruparelia, 2010). The history of version management systems started in 1972 with the Source Code Control System (SCCS) (Rochkind, 1975). The SCCS file format introduced the interleaved data storage technique, which is still used by some of the version control systems (e.g. BitKeeper). In 1980 Walter F. Tichy has developed a Revision Control System (RCS) which stored the revisions of the files under control using the UNIX “diff” command, which enabled version control for source code, documentation, procedural graphics, papers, and form letters (Tichy, 1982). Most of the later version control systems in the area of software engineering follow a similar approach: differences between text files are stored on a line basis and used for version control. The main drawback of RCS was that it was only capable of tracking the revisions of single files. This issue was addressed in the Concurrent Versioning System (CVS) by introducing a client/server model, support of branch imports and unreserved checkouts. Lately, other subversion systems have been developed (e.g. Git or BitKeeper) that allow for distributed version control (Ruparelia, 2010).

As the version control systems in software engineering are only applicable to text-based files, it is difficult to directly apply the same technologies to process models. Consequently, several researchers have published initial work on version management for conceptual models (Zhao and Liu, 2007; Küster et al., 2010; Thomas, 2008; Ekanayake et al., 2011). Most of these papers describe novel approaches for version management and highlight the uniqueness of representation and storage format of process models.

In 2007, process model version management was addressed with a narrow focus on workflow management systems (Zhao and Liu, 2007). The approach uses so called version preserving directed graph models (VPG) and presents a set of run-time modification operations. The intention of the method is to combine all changes to nodes and arcs together in one workflow model, depicted as a directed graph. On request, the desired version can dynamically be derived. For this purpose the traditional definition of directed graphs is extended by adding a set of version numbers, mappings between version numbers and nodes, and binary relations to represent exclusive relations between arcs.
in the VPG. For such a formal approach it is essential to specify modification operations like *add*, *delete* and *edit* in a formal manner. Based on this specification possible solutions to the problems of concurrent changes and conflict resolution are addressed.

In (Küster et al., 2010), an approach for applying change operations without a restricted order is presented. For this purpose, the concepts of partially and fully specified change operations are introduced. Moreover, the way to derive a fully specified change operation from a partially specified one with the help of graph transformation is presented. Based on these concepts, potential dependencies and conflicts of change operations are studied.

(H. Bae et al., 2007) describe a graph-based approach with check-in/check-out functionality that is similar to the approach by (Küster et al., 2010). In order to reduce storage size, the model is stored as a tuple of a root version and change operations, which are used to derive the most recent process model version.

Another arising problem is the size of the process repository. When trying to store all possible versions as an incremental growing model, the repository grows enormously fast. In (Ekanayake et al., 2011), an approach based on dividing the process model into independent fragments using Refined Process Structure Trees (RPST) is presented. The independent fragments help to reduce the repository size, as the process models tend to consist of similar process fragments. Furthermore, flexibility in concurrency model development is achieved because instead of the whole model, only a small part of the process is locked for editing.

Ideas on versioning management have also been developed in the context of reference models (Thomas, 2008). The authors claim that model versioning of incremental model development has been neglected, while research was focused on reference model variant management at the same time. The main result of the research is a high level architecture for the management of different versions of reference process models. The architecture consists of several enterprise databases, external applications, user interface elements etc. and interfaces for communication between them.

As an example of version management implementation, the Signavio process modelling tool (Signavio GmbH, 2012) can be taken into consideration. When a model is altered, it is automatically saved as a revision with comments on the changes made. As for the further practical implementation of version management in business process management, (Ekanayake et al., 2011) claim that the currently existing process modelling tools support only basic version control functionality on the level of process models or single process elements, not addressing the problems of concurrency control or conflict resolutions.

While the approaches for versioning conceptual models, as described above, use highly complex, proprietary concepts for model storage and versioning, we propose to apply generic concepts from the area of software engineering. These techniques are more mature and have been successfully applied in practice. As (Ekanayake et al., 2011) mention, conceptual models can be represented as XML files, which allows to apply simple comparison functions for text files (textual deltas). The approaches described by (Rochkind, 1975; Tichy, 1982; Ruparelia, 2010) can therefore be applied, once the conceptual models have been serialized to XML. We however go one step beyond textual deltas by exploiting the tree characteristic of XML documents. This allows to apply very efficient versioning techniques for tree structures (Choi and Kwon, 1997).
4 Versioning Concept

As a result from the literature review, we propose a combination of two different, in our view yet complementary, approaches for version management in process modeling: serialization and tree based versioning technique (Choi and Kwon, 1997; Mohamed et al., 2008).

4.1 Serialization

As motivated above, an important aspect of code versioning in software engineering is serialization. Source code is written with the help of more or less simple text files. Their content can easily be compared and versioned with the help of the two concepts: unit of versioning and unit of comparison.

The unit of versioning is an atomic entity. As soon as a part of this unit is changed, a new version is created. The commonly used unit of versioning in software engineering is a source code file or document. When choosing the unit of versioning there is always a trade-off between the possibility to trace committed changes and the number of created versions, which is directly related to the storage space required. On the one hand, with a fairly abstract unit of versioning that incorporates a high amount of content, it becomes complicated to trace committed changes. On the other hand, a very detailed unit of versioning with only a small portion of content will lead to more new versions during editing and therefore require more storage space.

![Figure 2. Typical unit of versioning and unit of comparison in software engineering](image)

The unit of comparison determines whether overlapping changes result in a conflict or not. As soon as a difference in the units of comparison between two versions is detected, the conflict has to be resolved.

The common unit of comparison in software engineering is the line of code. As Figure 2 depicts, there are other possible units for versioning and control within text-based documents, namely paragraph and word. Furthermore, those units are hierarchically ordered. In relation to this hierarchy it becomes clear that the process model scenario is more challenging since process models are usually not stored in text files but rather in binary or, in the best case, XML files. Therefore, the definition of the unit of comparison and unit of versioning cannot trivially be adopted. In any case, a serialization of the models is mandatory. Hence, a storage format has to be developed which is capable of preserving the possibly complex interrelations between the process elements including all process activities and their respective attributes. We therefore propose to use an XML-based storage format, which is capable of representing a complex, nested process landscape with its interdependencies and complementary attribution for each process within an all-embracing process framework. Although there have been efforts to standardize process model XML representations (Mendling and Nüttgens, 2006; OMG, 2012) there are still numerous proprietary formats and tools that do not support any exchange standards on the market. For demonstration purposes, we therefore use a simplified XML structure as shown in the following listing:
Within the listing above, a sample process landscape is defined by the XML storage format. Each process element has an identification number and a named XML attribute. By this means, the information connected to the element is saved only once even if the elements are used in several different locations. Hence, redundancy is highly reduced and consistency is fostered. This is especially true, when attributes are associated to the process elements (which are omitted in Listing 1 to improve readability). The element framework and its child elements represent the actual process structure. Each process element has an attribute that links to the respective definition and a unique object identifier (OID). The XML element edge links two detail processes or process building blocks via their identifiers. Figure 3 depicts the schematic process landscape of the XML example in Listing 1. The dotted lines indicate the demonstration of a more detailed specification on a subsequent layer starting from the process framework encompassing all main processes down to the atomic process building blocks on the most detailed layer.
4.2 Tree-based Versioning

The XML-structure outlined above can be represented as a tree. The root of the tree is the root element of the XML representation. In our example, this is the “<Project>” element. Child elements of the XML elements are represented as children of the parent node, e.g. “<Definitions>” and “<ProcessFramework>” as children of “<Project>”. It should be noted that the tree representation of the XML file preserves the structure and it can contain an arbitrary number of attributes. With the help of the XML-file and the tree-based approach the logical choice for the unit of versioning appears to be on the node level of the tree. Hence, whenever a (sub-)element within the file changes, a new version is created. However, there are basically three different approaches to create a new node in the tree structure (Choi and Kwon, 1997; Mohamed et al., 2008).

The first and most straightforward approach, called path-copy, is to copy every node from the root to the changed node as depicted in Figure 4. Whenever a node is changed in the tree structure, the whole structure until and including that changed node is copied and a new unique version number is created. In order to navigate to a certain version the user would have to choose or restore the corresponding root element.

The approach has a downside as the versioning effort increases dramatically in complex trees with long paths, which may occur in complex process landscapes. Furthermore, most nodes would be versioned and therefore stored duplicated although they did not change.

An improved approach, called Fat-Node, assigns new versions only to nodes which actually changed. The changed node is extended by a tuple, which determines the distinct versions. Obviously, the versioning effort in comparison to the copy-path approach is drastically reduced. On the downside, it
is more complex to display a certain version, because all previous versions have to be traversed. In Figure 5 there is a second version for node B, which is not active in the example.

![Figure 5. Fat-Node approach (Choi and Kwon, 1997)](image)

**Version-Controlled tree** is the third and most elegant approach to tree versioning, which combines the advantages of the aforementioned approaches. It allows for simple version selection for a whole (sub-) process structure and there is no node redundancy in the respective versions. This is achieved by the usage of so called history nodes. As displayed in Figure 6, the history node $h_i$ is connected to the parent node A if node B is changed. It contains the information of the version $i$ and the information about the child nodes of node A in this version. In case a specific version has to be retrieved, each node has to be checked for a history node which matches the desired version number $i$. The main difference to the Fat-Node approach is that the complementing version information is not appended to the node itself but stored in a separate history node. The advantage is that several nodes can be changed and this information can be saved in one history node. Using the example in Figure 5, additionally to the changes in node B also node C and D could have been changed in version $i$ and one history node would suffice to store all relevant information for the version $i$.

![Figure 6. Version-Controlled approach (Choi and Kwon, 1997)](image)

The most recent version in this approach is always directly displayed. The history nodes are only accessed, when a previous version of the model is required. Since only the changed nodes are saved in a history node, the versioning effort and the access time even in complex process landscapes is relatively low in comparison to the Fat-Node and Copy-Path approach (Choi and Kwon, 1997). The proposed versioning concept is therefore able to version even nested process models, as depicted in Figure 7.
Each node is logically linked to one level of abstraction, namely main processes, detail processes and process blocks. Each node holds a set of information. The most basic information is the name of the node and what child elements the node possesses.

As mentioned above the logical choice for the unit of versioning is a node of the tree representing one business process model. For example in Figure 8, the detail process “Handle Invoice” was renamed to “Accept Invoice”. According to the Version-Controlled tree approach, a history node \( h_1 \) was created. The history node stores the link to the old version of the detail process, namely “Handle Invoice”.

The most recent version of a model is retrieved, when the history node is disregarded. Only to return or to compare with older versions the history node has to be utilized. Furthermore, new versions are only created for changed nodes, to reduce the versioning effort and runtime in comparison to the two other presented approaches (Choi and Kwon, 1997).
4.3 Tool Requirements for Collaborative Model Versioning

In order to implement the approach described above, the modelling tool has to fulfil a certain range of additional requirements. This holds especially true in an assumed scenario of timely and locally distributed modelling (Hadar and Soffer, 2006; Batini et al., 1986).

First of all, rights management and authentication methods are highly recommended for a versioning system to work. It allows controlling if and to which extent modifications to all models or just to a subset of them are allowed for a specific user. In order to differentiate between the users, a way of authentication is mandatory. In case of a modern web-based modelling tool the model repository is stored online and also the working copy of the modeller is changed online. Hence, there is no sophisticated protocol needed to interchange the model operations between user client and server repository. Nevertheless, the atomicity of the single changes is important. Every commit and update of a model has to be handled in a transactional manner. Either the action is executed completely and stored in the repository or not at all in order to avoid inconsistencies.

To make full use of the versioning possibilities an important requirement is the possibility of branches. The support for branches can be manifold. For example, Subversion (Apache Software Foundation, 2012) allows applying model modifications to several branches or to merge branches. To support branching in our concept, nodes could be assigned to branches. When retrieving the actual and historic versions of the model, only nodes and history nodes of the selected branch have to be selected.

Since a versioning approach does not necessarily hinder conflicts to occur, the tool has to support the modeller with conflict resolution. Similar to current software engineering practices, we intend to use a three-way-merge approach (Lindholm, 2003). Herby, the different model versions, e.g. client and repository model or model versions of different users, are compared and merged. When resolving a conflict, the modeller is presented with both versions and can decide which aspects to keep and which to discard.

5 Conclusion

This paper presents a concept to adopt mature versioning approaches from software engineering to the area of business process modelling. Our proposed approach combines an XML-based storage format with a graph theory based version management on the tree structure of the XML document. The approach is demonstrated using a real-world example of a nested business processes landscape. The results disclose that the concepts cannot be transferred directly without further alignment but that a combination of software engineering approaches and graph theory provides a considerable solution. By deriving additional requirements for an implementation of the concept, we show the approach to be promising and outline that a sophisticated solution for versioning in the area of process management is possible.

Although only three layers of process nesting are used for demonstration purposes, both XML storage format and versioning concept is applicable for arbitrary levels of nesting due to the tree structure of the information. Naturally, we assume the concept to be applicable to all types of conceptual models that can be represented and stored using an XML tree structure. Judging from the ability of most process modelling tools to export models as XML files, this includes all commonly used modelling languages.

Nonetheless, the XML representation of the model only acts as a proxy to enable efficient versioning techniques. To eliminate this step, the process model itself could be regarded as a graph (Becker et al., 2012). Conceptual models created in languages such as ERM, EPC or BPMN are however prone to resemble graph structures more complex than trees, although a high percentage of these models have empirically been demonstrated to resemble trees (Breuker et al., 2012). Future research could therefore cover an extension of the tree-based versioning concept to complex graph structures, e.g. arbitrary networks.
Besides an extension of the concept we intend to implement the approach within a process modelling tool, as outlined in section two. We believe that especially the integration of our approach with non-relational database technologies such as document-oriented or graph-oriented databases will prove to be fruitful. Once implemented, we aim to evaluate both the feasibility and efficiency of our artefact(s).

References


