Focused Identification of Process Model Changes

Martín Soto and Jürgen Münch

Fraunhofer Institute for Experimental Software Engineering
Fraunhofer-Platz 1
67655 Kaiserslautern, Germany
{soto,muench}@iese.fraunhofer.de

Abstract. Advanced software process management requires capabilities to systematically analyze differences between versions of a process model. These capabilities can be used, for instance, to support process compliance management, to learn from process evolution, or to identify and understand process variations in different development environments in order to develop generic process models such as process standards. Analyzing the differences between process models versions is a highly challenging task that needs to be based on appropriate methods and tools. Experience has shown that, beside global version comparisons, local and focused difference analyses are often needed. Example goals of such focused analyses are the identification of all process changes that are relevant for a specific role, or the identification of those process changes that are relevant for a process reassessment. This article presents a technique based on pattern-matching for such focused analysis. The technique is a component of the comprehensive *DeltaProcess* approach for difference analysis [1, 2]. We explain the underlying concepts of the technique, describe a supporting tool, and discuss our initial validation in the context of the German V-Modell XT process standard. We close the paper with related work and directions for future research.

Keywords: process modeling, process model change, process model evolution, model comparison.

1 Introduction

Process engineers working on the evolution and maintenance of process models often have the need to compare different versions of these:

- When a new release of a process standard is published, users of the standard (e.g., organizations basing their own processes on the standard) need to know what changed, in order to adapt their own models.
- When a company-wide process model is modified, employees working according to that process need to determine if they are affected by the changes.
- When a model that involves safety or mission-critical aspects is updated, reviewers and auditors can work better and more efficiently if they know exactly what was modified.
- When simultaneous variants of a model are maintained (e.g., by several projects in an organization, or by several companies adopting and tailoring a single process

Q. Wang, D. Pfahl, and D.M. Raffo (Eds.): ICSP 2007, LNCS 4470, pp. 182–194, 2007.

framework), it is useful to be able to synchronize changes between variants, which requires finding out exactly what was changed in each variant.
– When company-specific standards are originally defined, or when they evolve, it is necessary to understand the process variability, both over time as well as across the organization's project space.

One frequently asked question is whether determining where changes occurred and why is just a matter of proper change control: it could be argued that logging every change made is all that is needed. In practice, however, this is not as easy as it may appear. Maintaining change logs is tedious and difficult, and documenting the changes is often seen as unnecessary overhead to more important tasks, like actually working on improving the process model. For these reason, change logs are often missing or poorly maintained and, thus, unreliable as a source of information.

There are many other situations in which mechanisms for reliably determining the differences between process model versions can be useful. These mechanisms must be focused, and deliver results that different process stakeholders (process engineers, project managers, developers, etc.) can use for different purposes. Research-wise, process difference analysis is faced with many challenges, such as 1) defining appropriate notations for specifying difference analysis goals, 2) creating suitable comparison algorithms, and 3) designing purpose-and stakeholder-oriented presentations of the results. The work we are presenting here is part of the *DeltaProcess* approach to software process model difference and evolution analysis [1, 2]. This paper concentrates on one single aspect of the approach, namely, given a particular user's information needs, how to identify precisely those changes that provide the user with the needed information. The technique presented here for this purpose comprises three main phases: In the first phase, the models are converted into a representation that makes it easier to compare them, since it regards all types of changes uniformly. In the second phase, a so-called comparison model is produced, which comprises the contents of both compared versions. In the third phase, focused difference analysis goals are specified, and a pattern-matching system is used to look for corresponding instances in the comparison model, and produce analysis results.

The rest of the paper is organized as follows: Section 2 describes the process model comparison problem using an example to illustrate the difficulties involved. Section 3 presents our change identification technique in detail, and includes some examples of its application to common, practical situations. In Section 4, we briefly discuss our implementation of this technique, as well as our experience applying it to the German V-Modell XT [3] process standard. The paper closes with Section 5, which compares our approach to related work, and Section 6, which presents our final conclusions and outlook.

2 The Process Model Comparison Problem

To illustrate the difficulties involved in process change identification, Fig. 1 shows two revisions of a process model excerpt, which we kept deliberately small for the purposes of the example. If someone were commissioned with the task of finding *all* differences manually (i.e., by looking at the diagrams) it would probably take this person some time to find all of them, and to make sure that none is missing, however

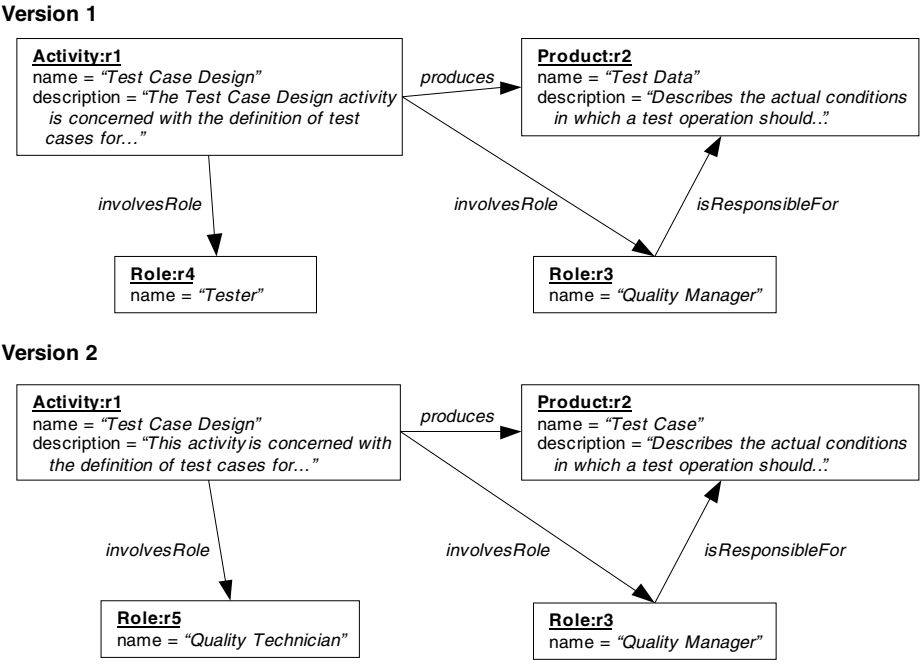

Fig. 1. Two versions of a process model (UML object diagram notation)

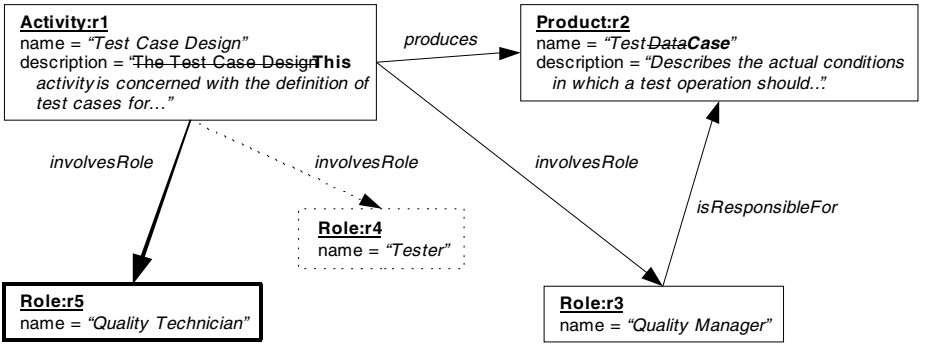

Fig. 2. A comparison of the process model revisions shown in Fig. 1

small it may be. Considering that real world process models are significantly larger than our example, it is clear that automated support is necessary to guarantee consistent and reliable comparison results.

Fig. 2 is an attempt to display all changes in the context of the changed model. Entities and relations erased from version 1 are marked using interrupted lines, whereas entities and relations new in version 2 are marked with thicker lines. Changes in entity attributes are shown by crossing deleted text off, as well as displaying new text in bold type. The following are some observations about this model comparison:

- *Changes are heterogeneous.* A number of basic change types can be directly identified from the example. They include entity additions and deletions, relation additions and deletions, and changes in the values of the various attributes.
- *Attribute changes must be interpreted according to model semantics.* Not all changes belonging to one of the basic types listed above are the same for the user. For example, if an attribute containing an integer value changes, it is probably enough to provide the user with the old and new values. On the other hand, if an attribute contains text changes, it is potentially important to determine which words were modified.
- *Relation changes must be interpreted according to model semantics.* For example, if a parent-child relationship changes, it is important to tell the user that a certain object has a new parent. If a consumes-produces relationship changes, it is important to report that some activities now produce new products, or that some products are now consumed by new activities. How this is reported may even depend on the role of the user with respect to the process. Oftentimes complete sets of simple structural changes must be grouped together and presented to the user as a unit for proper interpretation.
- *A graphical display is not enough.* Although Fig. 2 does a fairly good job of making changes obvious, the same type of display would not work if applied to a model containing hundreds or even thousands of entities. Even if the technical difficulties of producing such a large graph were overcome, finding all changes relevant to a particular task would still be difficult because of the sheer size and complexity involved.

3 Pattern-Matching Based Change Identification

In the following, we discuss our technique for model change identification. This technique makes it possible to handle a wide variety of types of changes in a completely uniform way, to flexibly define the types of changes that are considered interesting or useful (this can be based on the structure and semantics of the metamodel), and to restrict the results to only certain types of changes, or even to certain interesting portions of a model.

3.1 A Normalized Representation for Process Models and Their Comparisons

Our first step consists of representing models (and later their differences) in such a way that a wide range of change types can be described using the same basic formalism. The representation we have chosen is based on that used by RDF [4] and similar description or metadata notations. For our purposes, this notation has a number of advantages over other generic notations:

- Being a generic notation for graph-like structures, it is a natural representation for a wide variety of process model schemata.
- It has a solid, standardized formal foundation.
- As shown below, the uniformity of the notation, which does not differentiate between relations and attributes, makes it possible to describe a wide range of changes with a straightforward pattern-matching notation.

– Also as shown below, the fact that many model versions can be easily put together into a single model makes it possible to use the same pattern-matching notation for single model versions and for comparisons.

Fig. 3 shows the first revision from Fig. 1 converted to this representation. The graph contains only two types of nodes, which we will call *entity* nodes (ovals in the figure) and *value* nodes (boxes in the figure). Entity nodes have arbitrary identifiers as labels. Value nodes are labeled by the value they represent, which can belong to a basic type (string, integer, boolean, etc.)

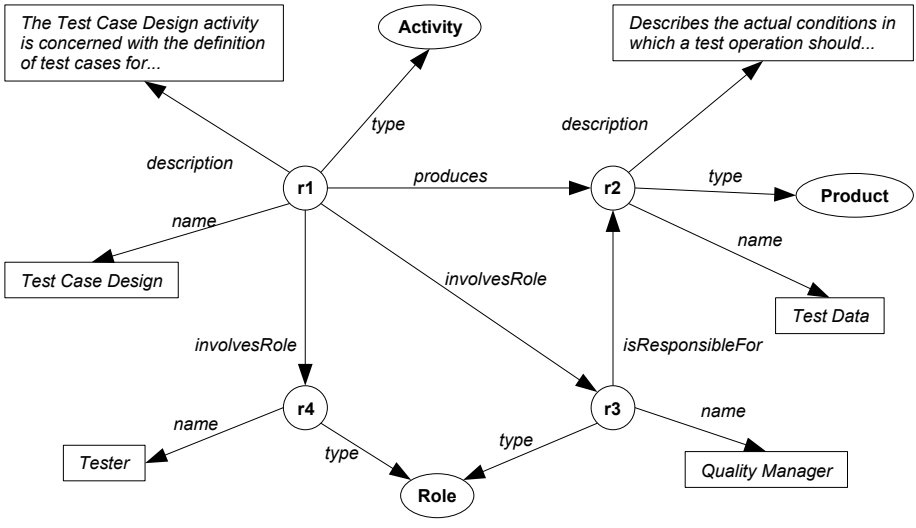

Fig. 3. A process model in normalized form

Arrows represent typed directed relations (type is given by their labels). Relations may connect two entity nodes, or may go from an entity node to a value node. It is not allowed for a relation to leave a value node. It is also not allowed for a node to exist in isolation. All nodes must be either the start or the end point of *at least* one relation. It follows that the graph is characterized completely by the set of the relations (edges) present in it, since the set of nodes is exactly the set of all nodes that are the start or the end of an edge.

The correspondence between attributed graphs (like those in Fig. 1) to this normalized form is straightforward:

– *Entities and types correspond to entity nodes.* For each entity instance and entity type in the original graph, there is an entity node in the normalized graph. There is also a *type* relation between each node representing an entity and the node representing its type.
– *Attributes correspond to entity-value relations.* For each entity attribute in the original graph, there is a relation labeled with the attribute name that connects the entity with the attribute value (that is, attributes in the original metamodel are converted into relation types). The value is a separate (value) node.

– *Entity relations correspond to entity-entity relations.* For each relation connecting two entities in the original graph, a relation connecting their corresponding entity nodes is present in the normalized graph.[1]

Fig. 4 shows the same comparison presented in Fig. 2, using the normalized notation, with changes also highlighted using interrupted and bolder lines. Formally, this graph respects exactly the same restrictions as the normalized model representation. The only addition is that edges are *decorated* to state the fact that they were deleted, added, or simply not touched. This leads us to the concept of a *comparison graph* or *comparison model.* The comparison model of two normalized models A and B contains all edges present in either A or B, or in both, and only those edges. Edges are marked (decorated) to indicate that the edge is only in A, only in B, or in both of A and B.

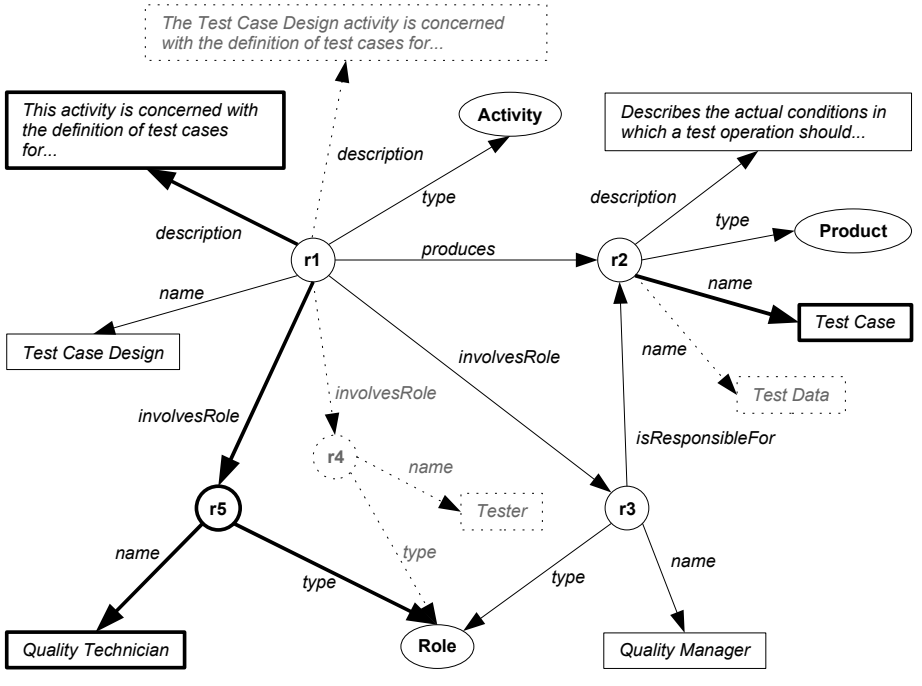

Fig. 4. A process model comparison in normalized form

The main aspect to emphasize here is the fact that all changes are actually reduced to additions and deletions of relations between nodes. This results in part from the fact that attributes are represented as relations, but also from the fact that nodes cannot exist in isolation. It is possible (and safe) to identify entity additions and deletions by looking for additions and deletions of *type* relations in the model. Also, to be fair, the comparison in Fig. 4 ignores one important aspect of Fig. 2, namely, the word level text comparison. We will deal with this limitation later in the paper (see Section 3.4).

[1] Relations with attributes can be modeled by introducing entity nodes that represent them, but the details are beyond the scope of this paper.

It is also important to point out, that the comparison model is useful only when entities have unique identifiers that remain stable after the model is changed. Although, in theory, this could be considered a strong limitation, it is seldom a problem in practice, since most practical modeling tools indeed provide unique, stable entity identifiers.

The fact that the normalized representation reduces all changes to sets of relation additions and deletions permits to describe many types of changes uniformly. The following sections discuss the mechanism that we have chosen to describe such changes in a clear and unambiguous way: a graphical pattern-matching language.

3.2 Graphical Comparison Patterns

In order to identify changes of a particular type, a so-called *graphical pattern*[2] must be defined, that matches precisely these changes. Graphical patterns are essentially comparison graphs in which some node and/or edge labels have been potentially replaced by variables. Informally, matching the pattern to a comparison graph implies replacing the variables in the pattern with actual labels from the graph, in such a way that the resulting pattern is a subset of the comparison graph. A value assignment for the variables in a pattern that results in a match is called an *occurrence* of the pattern. Normally, we are interested in the set of all occurrences of a pattern in a particular comparison graph. The following sections present examples of how to use this pattern language to identify interesting changes in process models.

3.3 Example 1: Additions and Deletions

Our first example is related to one of the simplest possible model changes: adding or deleting process entities. Fig. 5 shows four patterns that identify changes of this type with different levels of generality. The pattern in Fig. 5a) matches all additions of process activities, and for each match, sets the *?id* variable with the identifier of the new activity. In a similar way, the pattern in Fig. 5b) matches all deletions of process products. These patterns can be generalized to identify arbitrary additions and deletions:

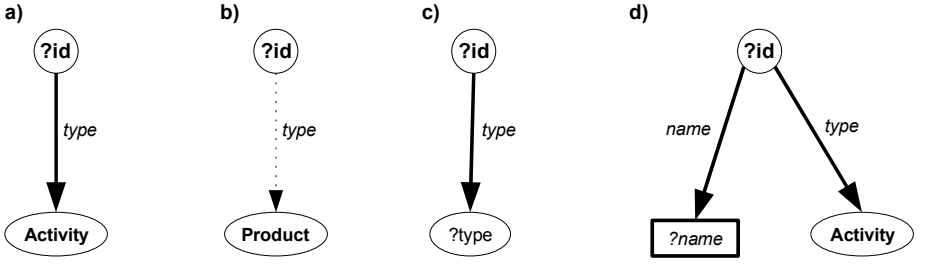

Fig. 5. Patterns for identifying entity additions and deletions

[2] Notice that our use of the word *pattern* is rooted in the standard pattern matching tradition, as related to problems like Prolog-style unification, term rewriting and regular expression search. It is not intended to relate to other uses of the word in computer science, like, for example, software design patterns.

the pattern in Fig. 5c) identifies all entity additions, and instances an additional variable with the type of the added entity. Finally, Fig. 5d) shows a pattern that not only finds new activities, but sets a variable with the corresponding name, a useful feature if the results of matching the pattern are used, for example, to produce a report.

3.4 Example 2: Changes in Attribute Values

Just as important as identifying entity additions and deletions, is finding entities whose attributes were changed. Fig. 6 shows three patterns that describe changes in attribute values. Fig. 6a) is basically an excerpt from the comparison graph in Fig. 4, which captures the fact that an attribute *description* was changed. This pattern, however, matches only the particular change shown in the example. The pattern in Fig. 6b) is a generalized version of the first one. By using variables for the entity identifier, as well as for the old and new property values, this pattern matches all cases where the *description* attribute of an arbitrary entity was changed. Notice that each match sets the value of the *?id* variable to the identifier of the affected entity, and the values of *?oldValue* and *?newValue* to the corresponding old and new values of the *description* property. The pattern in Fig. 6c) goes one step further and uses a variable for the attribute labels as well, which means it matches all attribute value changes. Notice that these patterns match once for *each* changed property in *each*

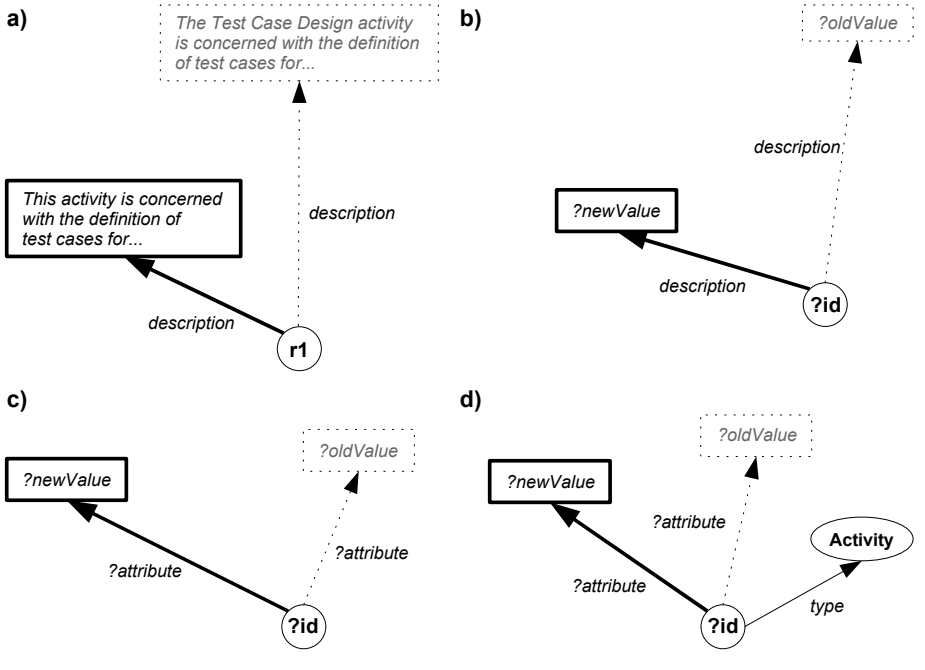

Fig. 6. Four patterns for identifying attribute value changes

object. Finally, the pattern in Fig. 6d) constitutes a specialization of its peer in Fig. 6c): it is restricted to all attribute value changes affecting process activities.

Changes identified in this way, can be fed to additional algorithms that perform attribute specific comparisons, like, for example, identifying added or deleted individual words or characters in text based attributes. These way, potentially expensive specific comparison algorithms are only applied to relevant items.

3.5 Example 3: Impact of Changes on Tool Usage

Our last example stems from a question posed to the authors by a process engineer: "How can I determine the impact of process changes on the software tool infrastructure?" The goal of this process engineer was to find out if new software development tools (or tool licenses) had to be purchased, and, if so, which people had to receive them after the new process changes were implemented.

Fig. 7 shows two patterns that could help answer this question. These patterns introduce a new language element: an edge marked with a black point (like the edge from *?toolId* to *?toolName* in Fig. 7a) matches edges in the comparison graph without regard to decoration. This means that old, new, and common edges could be matched by such an edge, as long as the labels of the end nodes and the relation label match.

By using this feature, the pattern in Fig. 7a) is able to match tools having a new *requiresTool* relation to a tool. This happens regardless of whether the tool itself is new or existed before, but was not required by the activity. The pattern in Fig. 7b) is a variation of the previous one, which matches new activities and connects them to the tools they require. These and similar patterns can be used to determine which users will eventually require new software tool licenses, in order to buy and install them timely.

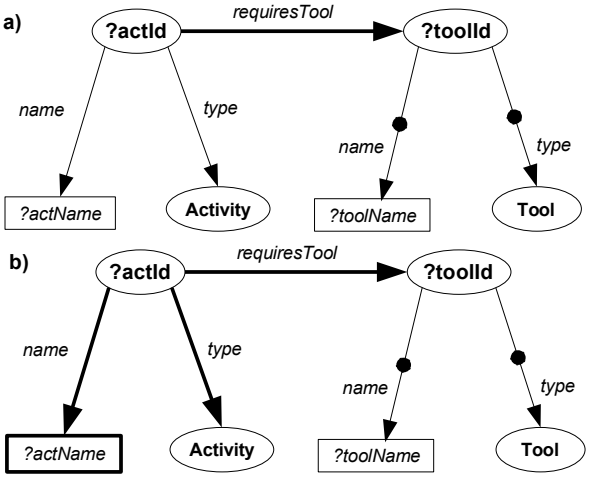

Fig. 7. Identifying the impact of activity changes on tool requirements

4 Implementation and Validation

An implementation of the pattern-matching based change identification technique presented in the previous sections is available as part of the *Evolyzer* tool (see Fig. 8), which is intended to support the *DeltaProcess* process model difference analysis approach mentioned in the introduction. We have tested our approach and tools by applying them to the various official releases of the V-Modell XT [3], a large prescriptive process model intended originally for use in German public institutions, but finding ever increasing acceptance from the German private sector. As of this writing, the *Evolyzer* tool still lacks a graphical editor for change patterns. However, patterns can be expressed as textual queries using a syntax that basically follows that of the emerging SPARQL [5] query language for RDF. Expressed as queries, patterns can be executed to find all their occurrences in a model.

The V-Modell XT constitutes an excellent testbed for our approach and implementation. Converted to the normalized representation defined in Section 3.1, the latest public version at the time of this writing (1.2) produces a graph with over 13,000 edges. This makes it a non-trivial case, where tool support is actually necessary to perform an effective analysis of the differences between versions. Our first trial with

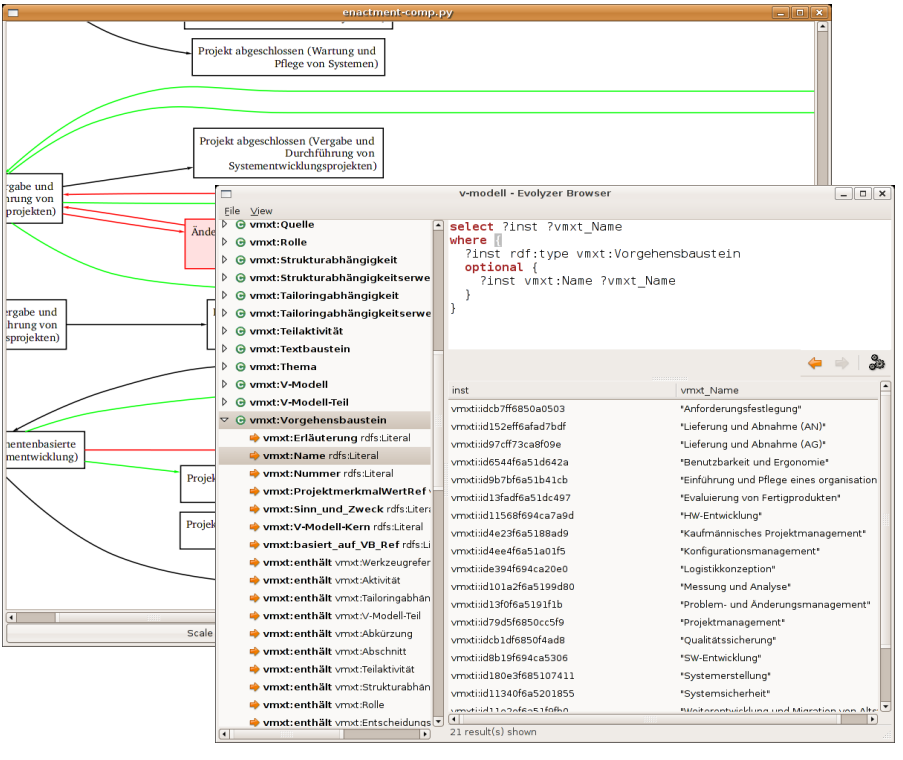

Fig. 8. The *Evolyzer* tool working on the V-Modell XT

the V-Modell XT consisted of analyzing the evolution of the “official” V-Modell XT itself: we compared the model's available public releases 1.0 (11,555 edges in our representation), 1.1 (11,822 edges) and 1.2 (13,286 edges). A comparison of versions 1.1 and 1.2, for example, yields 10,628 common edges, which indicates a common core of almost 80% of the latest version.

Using various patterns, we were able to classify the changes into groups, including added or removed entities, entities relocated inside the structure, renamed entities, corrected entity descriptions, etc. This analysis was motivated by the concrete needs of a company that has deployed a customized version of the VModell XT for use in all internal software development projects (Example 3 in Section 3.5 is also based on this work). One of the main purposes of the analysis is to use process model difference analysis to keep the company's customized model synchronized with the standard VModell XT.

Even for large cases like those just described, we consider the performance of our prototype implementation to be satisfactory. Running on a standard desktop PC, the system is able to convert an instance of the V-Modell XT into the normalized form in less than 10 seconds. Building the comparison model takes less than five seconds, and matching patterns is usually faster (of course, this depends on the complexity of the pattern). Most interesting differences can be analyzed interactively, which makes the system even more useful. Additionally, the Evolyzer framework makes it possible to feed changes identified by a pattern to further analysis algorithms. Currently is it possible to see changes in context using a graph layout system, and to produce text reports of certain types of differences.

5 Related Work

Several other research efforts are concerned in one way or another with comparing model variants syntactically and providing an adequate representation for the resulting differences.

[6] and [7] deal with the comparison of UML models representing diverse aspects of software systems. These works are oriented towards supporting software development in the context of the Model Driven Architecture. Although their basic comparison algorithms are applicable to our work, they are not concerned with providing analysis or visualization for specific users.

[8] presents an extensive survey of approaches for software merging, many of which involve comparison of program versions. The surveyed works mainly concentrate on automatically merging program variants without introducing inconsistencies, but not, as in our case, on identifying differences for user analysis.

[9] provides an ontology and a set of basic formal definitions related to the comparison of RDF graphs. [10] and [11] describe two systems currently in development that allow for efficiently storing a potentially large number of versions of an RDF model by using a compact representation of the raw changes between them. These works concentrate on space-efficient storage and transmission of change sets, but do not go into depth regarding how to use them to support higher-level tasks (like process improvement).

Finally, an extensive base of theoretical work is available from generic graph comparison research (see [12]), an area that is concerned with finding isomorphisms (or correspondences that approach isomorphisms according to some metric) between arbitrary graphs whose nodes and edges cannot be directly matched by name. This problem is analogous in many ways to the problem that interests us, but applies to a separate range of practical situations. In our case, we analyze the differences (and, of course, the similarities) between graphs whose nodes can be reliably matched in a computationally inexpensive way.

6 Conclusions and Future Work

Due to factors like model size and metamodel differences, the general problem of identifying and characterizing changes in process models is not trivial. By expressing models in a normalized representation, we are able to characterize interesting changes using a graphical pattern matching language. Graphical patterns provide a well-defined, unambiguous and, arguably, intuitive way to characterize common process model changes, as our examples show.

Our implementation of pattern queries in the *Evolyzer* system demonstrates that our pattern-based change identification technique can be used in practical situations involving very large process models like the V-Modell XT. It is important to stress, however, that the technique requires the process entities in compared models to have stable identifiers that are used consistently across versions. Thanks to the fact that common modeling tools support stable identifiers, this is often the case when comparing versions of the same model, but not when comparing models that were created independently from each other.

Acknowledgments. We would like to thank Sonnhild Namingha from Fraunhofer IESE for proofreading this paper. This work was supported in part by the German Federal Ministry of Education and Research (V-Bench Project, No. 01| SE 11 A).

References

1. Soto, M., Münch, J.: Process Model Difference Analysis for Supporting Process Evolution. In: Proceedings of the 13th European Conference in Software Process Improvement, EuroSPI 2006. Springer LNCS 4257 (2006)
2. Soto, M., Münch, J.: The *DeltaProcess* Approach for Analyzing Process Differences and Evolution. Internal report No. 164.06/E, Fraunhofer Institute for Experimental Software Engineering (IESE) Kaiserslautern, Germany (2006)
3. V-Modell XT. Available from http://www.v-modell.iabg.de/ (last checked 2006-03-31).
4. Manola, F., Miller, E. (eds.): RDF Primer. W3C Recommendation, available from http://www.w3.org/TR/rdf-primer/ (2004) (last checked 2006-03-22)
5. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Working Draft, available from http://www.w3.org/TR/rdf-sparql-query/ (2006) (last checked 2006-10-22)
6. Alanen, M., Porres, I.: Difference and Union of Models. In: Proceedings of the UML Conference, LNCS 2863 Produktlinien. Springer-Verlag (2003) 2-17

7. Lin, Y., Zhang, J., Gray, J.: Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In: OOPSLA Workshop on Best Practices for Model-Driven Software Development, Vancouver (2004)
8. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, Vol. 28, No. 5, (2002)
9. Berners-Lee, T., Connolly D.: Delta: An Ontology for the Distribution of Differences Between RDF Graphs. MIT Computer Science and Artificial Intelligence Laboratory (CSAIL). Online publication http://www.w3.org/DesignIssues/Diff (last checked 2006-03-30)
10. Völkel, M., Enguix, C. F., Ryszard-Kruk, S., Zhdanova, A. V., Stevens, R., Sure, Y.: SemVersion - Versioning RDF and Ontologies. Technical Report, University of Karlsruhe. (2005)
11. Kiryakov, A., Ognyanov, D.: Tracking Changes in RDF(S) Repositories. In: Proceedings of the Workshop on Knowledge Transformation for the Semantic Web, KTSW 2002. Lyon, France. (2002)
12. Kobler, J., Schöning, U., Toran, J.: The Graph Isomorphism Problem: Its Structural Complexity. Birkhäuser (1993)