A History Concept for Design Recovery Tools

Jens H. Jahnke
Department of Computer Science
University of Victoria
PO Box 3055, Victoria B.C.
Canada V8W3P6
jens@acm.org

Jörg P. Wadsack
Dept. of Mathematics and Computer Science
University of Paderborn
Warburger Str. 100
33098 Paderborn, Germany
maroc@upb.de

Albert Zündorf
Dept. of Mathematics and Computer Science
University of Braunschweig
Post box 3329
38023 Braunschweig, Germany
zuendorf@ips.cs.tu-bs.de

ABSTRACT

Many tools have been developed for recovering the design of legacy software. Interactively invoked abstraction operations and re-design transformations play a central role in these tools. A limitation of most existing approaches is, however, that they assume a mostly linear transformation process. They provide little support for iteration, recursion and incremental changes during the recovery process. Nevertheless, empirical results suggest that real-world abstraction and reengineering processes are in fact highly iterative. A history mechanism that explicitly maintains dependencies of all performed transformations can overcome this mismatch. Based on our experience with a specialized implementation of such a mechanism, we present a generalized history concept as an add-on to existing tools that support design recovery.

Keywords:

Software maintenance, design recovery, reengineering, reverse engineering, history concept, software transformation

1. Introduction

Over two thirds of today's expenditures in information technology are spent on the maintenance of existing software systems. Re-documentation and reengineering activities account for largest part of this budget. Many reengineering tools have been developed in industry and academia in order to support this task, e.g., [1-7]. Such tools provide mechanisms for creating conceptual abstractions of the analyzed software system. Typically, these abstractions are created in semi-automatic, human-centered processes. Interactively invoked abstraction and re-design transformations play a central role in these tools. However, case studies with industrial applications have shown that the applicability of current tools for human-centered design recovery is limited [8]. This is because they assume a relatively linear process with little support for process iteration, recursion and incremental changes during the reengineering process. The left-hand side of Figure 1 illustrates that reengineers typically loose large parts of the transformations performed manually when process iterations occur, i.e., when new analysis knowledge about the legacy system becomes available. Unfortunately, such iterations occur frequently in real-world abstraction and recovery processes [8].

A *history mechanism* that explicitly maintains dependencies among all performed transformations can tackle this problem. Such a history mechanism provides the tracability necessary for iterative reengineering processes. The right-hand side of Figure 1 illustrates this idea of an explicit dependency relation among all performed transformations. Let us assume that the black box represents knowledge about a software artifact that has been updated during the iteration. Then, the dependency information stored in the proposed history mechanism enables the selective removal of only those transformation operations that (transitively) depend on the changed information.

In [9], we presented a dedicated implementation of such a mechanism in our database reverse engineering environment. Based on our experience with this prototype, we now suggest a generalized history concept as an add-on to various kinds of existing design recovery tools. Of course, different tools have different application domains (e.g., object-oriented design, database schema design) but the vast majority of tools have in common that they are *graph-based*, i.e., they store model data in abstract syntax graphs (ASG). Therefore, our approach makes use of the theoretical concept of *graph-transformation systems* [10] as a suitable abstraction for all of these tools. This theoretical basis allows us to uniformly describe various kinds of design recovery processes in terms of graph processes [11]. Furthermore, the integration of our history mechanism with existing tools is facilitated by the current initiative for adopting a common graph-based data interchange format [12].

The rest of this paper is structured as follows. In the next section, we give an overview on related work in the area of document consistency preservation and history

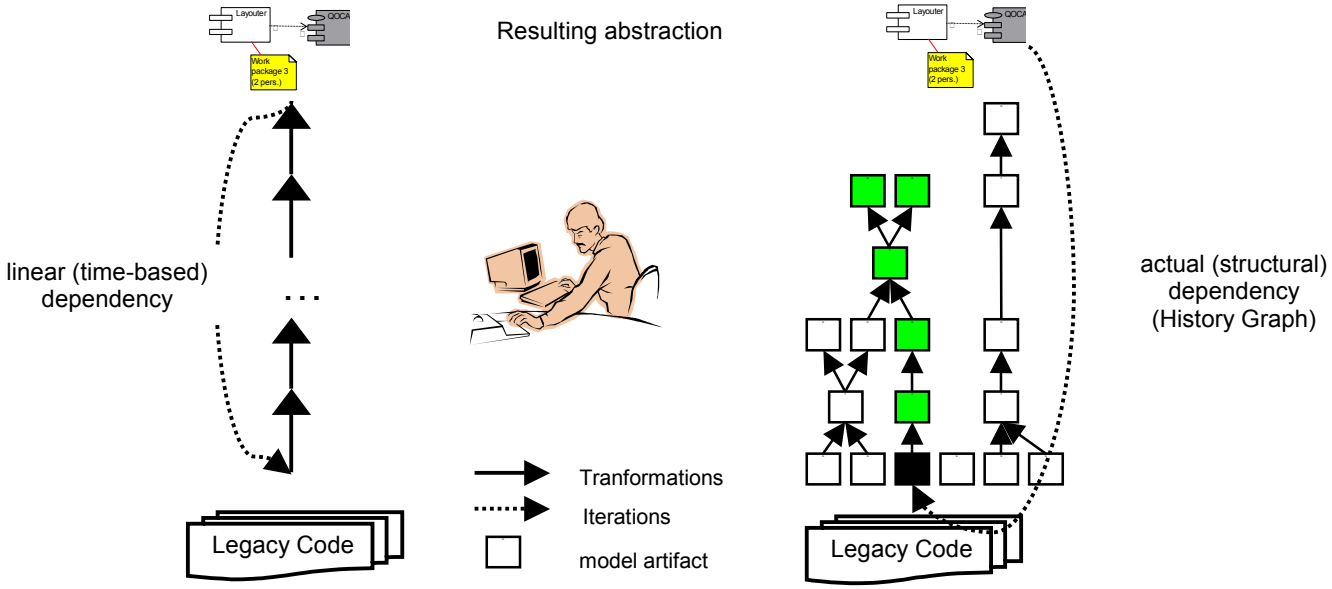

Figure 1. Iteration support without and with a History Mechanism for maintaining transformation dependencies

management. Section 3 introduces concepts that are fundamental for our approach, namely the concept of a graph and a graph grammar. Based on this definition, Section 4 outlines the proposed History Management mechanism. An application of this mechanism and its integration with an existing design tool is presented in Section 5. Finally, Section 6 closes with concluding remarks and future research directions.

2. Related work

The described research is related to various existing approaches to maintaining inter-document consistency in the software (forward) engineering process. Perhaps the most prominent technology in this regard is mechanisms for version and configuration management like e.g., the *concurrent versions system* (CVS) [13]. In principle, mechanisms like CVS can be used to create version histories of recovered design documents. However, such version histories do not consider the actual structural dependencies among the transformations performed in the abstraction process. Instead, the version history merely reflects a time-oriented view on intermediate stages reached in the (last iteration of the) recovery process. Consequently, traditional version management systems can only provide limited support for iteration as illustrated on the left-hand side of Figure 1.

Lefering and Schurr suggest a dedicated formalism called *Triple Graph Grammars* for specifying structure dependencies among Abstract Syntax Graphs (ASGs) [14]. This formalism is used in the IPSEN project for generating integration tools that maintain consistency and propagate changes from software design documents to the implementation code and vice versa [15]. Nowadays most commercially available development environments provide (limited) support for propagating changes between software documents on different levels of abstraction.

Unfortunately, very few reverse engineering tools provide similar support for propagating changes of the knowledge about a legacy system to its reverse engineered abstraction. Current reverse engineering tools assume a similar waterfall-like process that was imposed by forward engineering tools two decades ago. Typical human-centered reverse engineering tools (e.g., [1, 2, 16, 17]) provide extractor programs for populating the tool repository. The contents of the repository can then be visualized and transformed with various kinds of viewers and editors. With most current tools, these two steps (1) legacy system analysis and information extraction, and (2) information transformation and abstraction cannot be iterated without losing the interactive work performed on abstracting and transforming the information on the legacy system.

One notable exception is the DB-MAIN database reverse engineering environment developed by the University of Namur, Belgium [18]. DB-MAIN logs the history of all interactions during interactive abstraction and refactoring sessions in order to better support continuous evolution in relational databases. This allows the reuse of the operational "knowledge" accumulated during the different phases of a reengineering process. The DB-MAIN history mechanism addresses issues of

consistency maintenance during evolution of the persistent data (schemas) during the life cycle of a database application. Hick et al. have developed different "evolution strategies" depending on whether the evolution takes place in the physical, logical or conceptual schema. Each performed transformation is recorded precisely and completely to make its inversion possible. The history has to be monotone and linear, which implies no iteration and no branches. Concurrent work is not supported, i.e., only one person can modify the schema at the same time.

The DB-MAIN consistency management strategies depend on the reversibility property of the transformations applied. This property makes it possible that changes can be propagated from one schema level to another. In case that a transformation recorded in the history becomes invalid, i.e., its precondition fails, the reengineer has to resolve the inconsistencies manually. In contrast to the History Mechanism proposed in this paper, DB-MAIN does not allow the reengineer to get back (and modify) the initial state of the legacy system.

The *Varlet* database-reengineering environment is another tool that supports iterative analysis and abstraction processes [9]. In this approach, *Varlet* logs all database schema transformations invoked by the user in an automated logbook (including their pre- and post-conditions and interdependencies). Moreover, the initial, low-level representation of the legacy schema remains available for the user. This enables the user to revisit the starting point of his/her interactive modification at a later point in the reverse engineering process in order to modify existing or add new low-level information on the legacy system. The new information might stem from applying additional extractor programs or from interviews conducted with developers or other human experts. Whenever the initial representation of the legacy system is changed, Varlet uses the logged transformation dependencies to determine which interactive operations are affected by the modifications. The pre-conditions of these transformations are then re-evaluated and only those transformations that fail this test are undone. After this propagation step, the consistency of the extracted information on the legacy system and its interactively created abstraction has been reestablished.

This paper describes a generalization of the Varlet consistency management mechanism as a reusable component to be integrated with existing transformation-based reverse engineering tool. Compared to the original mechanism implemented in Varlet, the functionality of the consistency management component described in this paper is limited to determining and undoing *all* interactive operations that are affected by a change of the legacy system. This means that the preconditions of operations are not re-evaluated. The rationale behind this limitation is to minimize the requirements on tools that can interface with our component. The re-evaluation of the performed operations would require tools to explicitly disclose all checks (preconditions) performed prior to the execution of each operation. This information is normally not available.

3. Graph Transformation Systems

Graphs are commonly used by reverse engineering tools for internal representation of software artifacts, e.g., [1, 2, 16, 17]. Of course, the specific graph models used in different tools might comprise variations with respect to their expressiveness. Nevertheless, most graph models have in common that they support different node- and edge types, attributes and directed edges. Recently, the reverse engineering research community has developed a standard graph model (called GXL) to promote tool interoperability [12]. Since the start of this initiative in 1998, an increasing number of tools have been extended with GXL import / export functionality. In the following, we will use the GXL graph model as a basis for presenting our approach. Still, the reader should note that our technique does not depend on using this particular model.

The GXL Graph Model

In the following, we will use the UML for giving a brief, semi-formal introduction to the GXL graph model. We refer to [10] for a complete formalization of attributed graph models. A more in-depth discussion of GXL can be found in [12]. Figure 2 shows a UML specification of the GXL graph model.

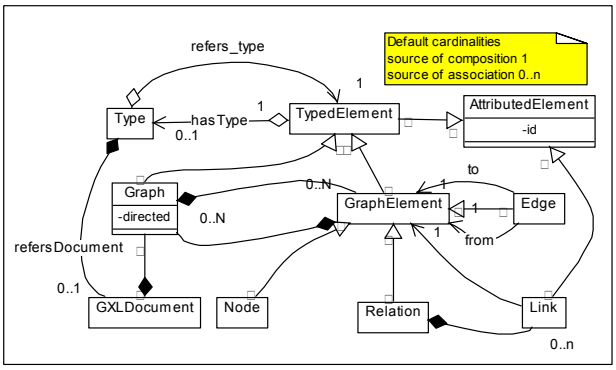

Figure 2. GXL graph model

A graph can be directed or undirected. It contains graph elements in form of nodes, edges, and relations. Relations contain a set of links pointing to one graph element each. All graph elements have unique identifiers.

GXL also includes the notion of hypergraphs, i.e., graph elements can themselves contain sub-graphs. Furthermore, GXL supports typed graph elements (where the type of a graph element is itself a graph element of a type graph). We omit a more detailed discussion of the GXL typing concept since it is not a required in this paper. Note that the actual exchange of GXL graph instances is performed in a canonical textual format based on XML [12].

Graph productions

Graph transformation systems (or graph grammars) are typically defined as a start graph and a set of graph transformation rules (or graph productions). Generally, graph productions can be defined as a pair of graphs, a set of *application conditions*, and a set of attribute *transfer clauses*. The two graphs are called the *left-hand side* and the *right-hand side* of the production, respectively. Graph elements of the left-hand side might also appear on the right-hand side. Graph productions and their application semantics have been formalized based on algebra theory. A complete formalization of these concepts is out of the scope of this paper and can be found in [10]. The intuitive understanding provided by the following semi-formal definitions is sufficient for the purpose of this paper.

Defintion 1. Graph production

A graph production is a tuple r:(P, Q, C, T), where

P(r)=P and Q(r)=Q are two graphs over the same sets of node and edge type labels;
P(r) is called the left-hand side and Q(r) is called the right-hand side of r.

C is a set of application conditions.

T is a set of attribute transfer clauses.

Defintion 2. Application of a graph production

*A production r:(P,Q,C,T) is **applied** to a host graph G in the following five steps:*

- ***choose** an occurrence of the left-hand side P in G. P has an occurrence in graph G if there is a morphism m:P→G which preserves source and target and labelling mappings.*
- ***check** the application conditions C. If they hold the occurrence of P in G is called a **match** for P.*
- ***remove** all elements in G that have been matched to elements in P that do not occur in Q,*
- ***adding** all elements to G that are new in Q. The occurrence of Q in the modified host graph G is called **comatch**.*
- ***transfering** attribute values to nodes in G that match nodes in Q(r) according to the attribute transfer clauses specified in T.*

We denote $G\downarrow^{(r,m)}$ for the graph that is produced by the application of a production *r* to a graph *G* (in a match *m*).

Note that graph productions are typically represented graphically, e.g., using the notation proposed by Schurr et al. [19]. In the following, we will discuss the sample graph production shown in Figure 3.

Operations viewed as graph productions

In principle, any tool for software design recovery can be viewed as specific graph transformation systems, where each modifying operation offered to the user is represented by a graph production. For example, consider a refactoring operation *generalize* that creates an abstract super class for a selected class *c* and generalizes some members of *c*. Figure 3 shows a representation of this refactoring operation in form of a graph production.

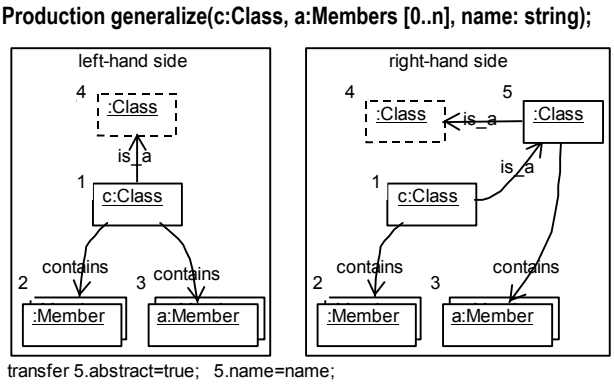

Figure 3. Graph production *generalize*

The unique identification number in the upper left corner of each node allows identifying identical nodes on both sides of the production. Node 1 represents the class to be generalized. Node 3 represents the set of members that will move to the new abstract super class (Node 5). Node 2 represents the set of all remaining class members[1]. Node 4 considers the case that the class to be generalized might already have a super class[2]. In this case, Node 4 will become the super class of the new generalization

[1] Note, that both nodes (2 and 3) are determined by the user's selection, parameter *a*.

[2] The dashed border of Node 4 specifies that a match for this node is optional. We adopted this notation from [19]. This notation is merely an abbreviation since productions with optional graph elements can always be presented by a set of productions without optional graph elements.

(Node 5). The transfer clause at the bottom of Figure 3 specifies that the new class is abstract by assigning the true value to the corresponding attribute of Node 5.

Creating Reengineering Histories with Graph Processes

The overall goal of our research is the development of mechanisms that facilitate incremental and iterative reverse engineering processes. *Tracability* is the most important prerequisite for such a mechanism, i.e., in case of process iterations, we have to be able to trace and propagate modifications of the initial representation of the legacy system to its transformed representation, in order to re-establish consistency. The requirement for tracability implies some sort of "logbook" about the interdependencies of all operations invoked by the user. We will now propose the realization of such a logbook based on the formal concept of *graph processes* [11].

Analogously to the treatment of operations as graph productions, we can view operation applications as applications of graph productions. In the previous section, we pointed out that the application of a graph production could be defined by its *match* and *comatch*. More precisely, the application of a production *r:(P,Q,C,T)* to a host graph G is defined as a tuple of morphisms (m:P→G, m':Q→ $G\downarrow^{(r,m)}$) where m denotes the *match* and m' denotes the *comatch*.

As an example Figure 5 shows an application of production generalize to the sample host graph in Figure 4. The left-hand side of Figure 5 shows the match while the right-hand side shows the comatch.

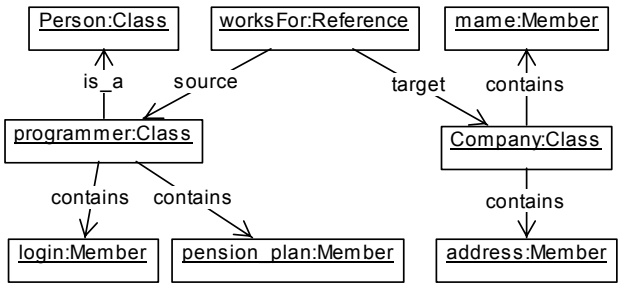

Figure 4. Sample host graph

Real-world reverse engineering processes involve numerous such operation applications invoked by the user, which might partly depend on results created by previous operation applications. A history mechanism has to keep track of these operation applications and their interdependencies. Again, we propose a graph structure for this purpose. The top part of Figure 6 shows the general form of such a graph-based representation of an operation application (called transformation for short). The bottom part of Figure 6 illustrates this representation for our example in Figure 5.[3]

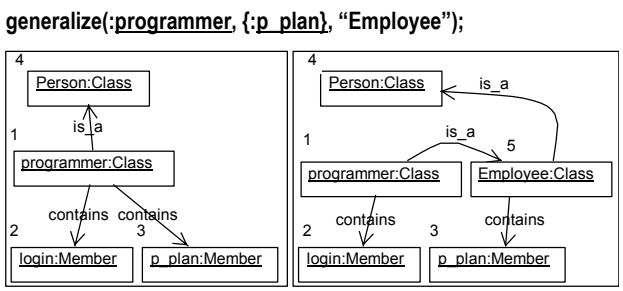

Figure 5. Application of production *generalize*

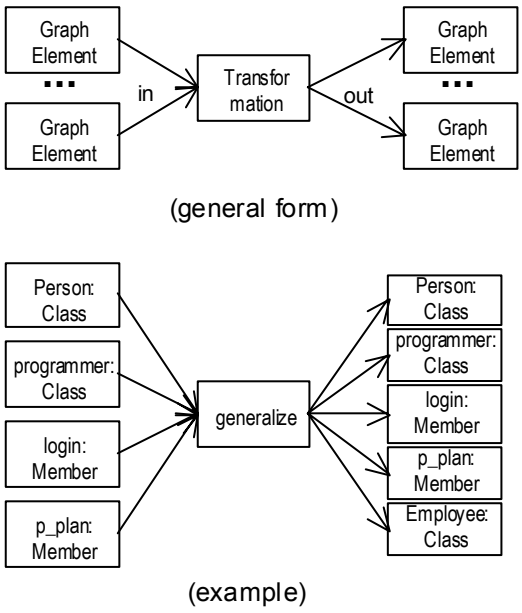

Figure 6. Graph representation of operation applications

We call the graph that stores the dependencies among transformations the *History Graph*. Figure 7 illustrates the structure of this graph, where T-nodes represent transformations and G-nodes represent elements of the Abstract Syntax Graph (ASG). Each transformation consumes certain ASG elements and replaces them by other ASG elements. In graph grammar terms, the consumed elements represent the *match* of the

[3] For readability reasons, we only listed graph nodes as input and output of transformation generalize. However, the internal representation of the History Graph maintains nodes and edges as input or output of transformations.

corresponding graph production, while the produced ASG elements represent its comatch. From the above discussion follows that the History Graph, which represents a particular software refactoring history, contains the current ASG of the refactored SW systems as a *subgraph*. This subgraph can easily be determined by filtering all graph elements that have not been consumed by transformations, i.e., that are not sourced by an '*in*' edge that points to a transformation node (bold G-nodes in Figure 7).

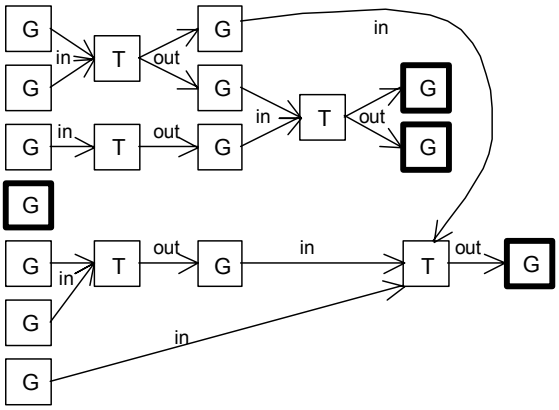

Figure 7. History Graph

Likewise, the History Graph also subsumes all previous states of the ASG during the interactive refactoring process. Thus, it is possible to trace back to any past state in the editing history.

The History Graph is a specific implementation of the general concept of a *graph process* as introduced by Corradini et al. [11]. Corradini defines a graph process as a partially ordered structure, plus suitable mappings which relate the elements of this structure to those of a given typed graph grammar.

4. Using History Graphs for consistency management in reengineering tools

We have employed the concept of History Graphs for consistency management during iterations in design recovery processes. For example, assume that midway during a reverse engineering process the reengineer learns about additional dependencies in the legacy system. Ideally, (s)he would like to add this new information to the initially extracted representation of the legacy system and validate those refactoring transformations which might have to be undone due to the new knowledge. The History Graph concept serves exactly this purpose. With its help the user can get back to the initial representation of the legacy system, make all desired changes, and determine which transformations might have lost their validity.

In the Varlet project, we developed a prototype reverse engineering tool that uses History Graphs for this purpose [9]. In addition to their input/output dependencies, transformations in the Varlet History Graph also log their pre- and post-conditions. This enabled us to perform the validation of transformations fully automatically. On the downside, the History Graph mechanism built in Varlet requires tight integration with the rest of the tool and cannot be reused in other existing tools. Therefore, we have developed a lightweight History Graph mechanism that can loosely be integrated with existing reverse engineering tools.

4.1 Tool integration prerequisites

A tool has to meet certain requirements to be able to interface with our History Graph component. We have minimized these requirements as much as possible to enable the integration with many different environments. Basically, a tool has to be able to

1. import and export its internal ASG structure (preferably in GXL format), and
2. for each design recovery transformation on the ASG
   - report the graph elements that represent the input of the transformation[4]
   - report the graph elements that represent the output of the transformation

   in GXL format.

An increasing number of current reengineering tools fulfill Requirement 1, e.g., Rigi [2]. Others can easily be extended because they are end-user programmable. For example, we have extended UMLStudio[5] in our sample application described in the next section. Requirement 2 is easy to achieve if Requirement 1 is satisfied because the input and output of a transformation can be represented as two GXL graphs (i.e., the match and the comatch according to Definition 2).

The sequence diagram in Figure 8 illustrates a typical interaction scenario between a design recovery tool and the History Management component. Note that the diagram covers a single iteration in the recovery process only. The process starts with the creation of the start graph by passing it to the `GXLHistory` component, (`addGraph(sG)`). Then, a sequence of transformations is performed by the user. These transformations (`t1` to

[4] Including all graph elements that are checked in checks of preconditions of the operation.
[5] www.pragsoft.com

tn) are logged in the History Graph. This is done by calling function addTransformation(l,r,t), where l and r denote in- and output of transformation t, respectively.

At any point in time, the user can inspect the editing history. This is done by invoking the createHistory() operation. As result (s)he obtains the list of the transformation t1 to tn with the respective time stamps.

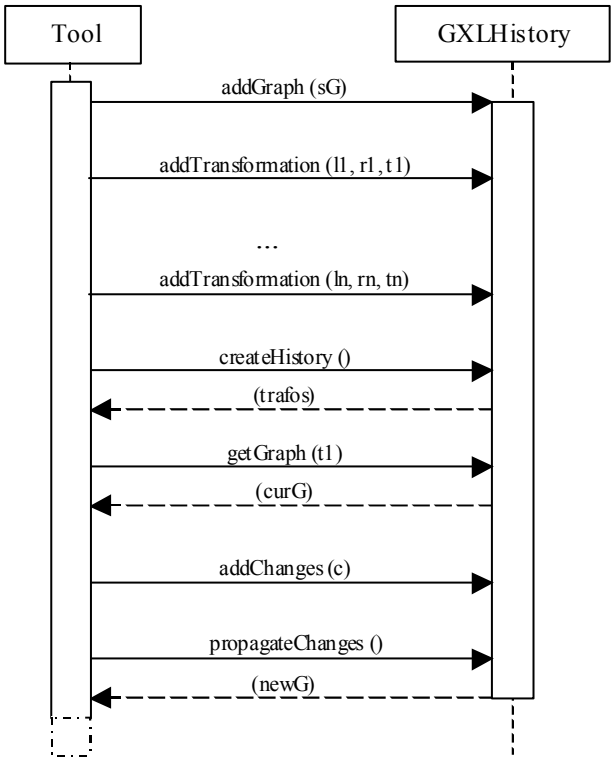

Figure 8. Interaction with History Component

When an iteration of the reverse engineering process is required, the tool restores the start graph by passing the transformation t1 to the GXLHistory component (cf. getGraph(t1)). Subsequently, the user can perform all necessary additions and/or modifications to the initial representation of the legacy systems and submit the changes to the GXLHistory (cf. addChanges(c)). Likewise, the user may restore any other situation by calling getGraph (tx) with an intermediate transformation tx and by editing that situation.

Now the User can use the logged history information to re-establish consistency through an incremental undo of all transformations that depend on the changes to the start graph (propagateChange()). As result of this operation (s)he gets the current graph ((newG)) without the undone transformation.

4.2 History Management algorithm

Internally, we use an extended version of the GXL graph model for maintaining the History Graph: we have introduced *transformation* nodes that carry a *timestamp* and a *name* attribute (cf. Figure 9). Transformation nodes are created automatically by the History component for each call to addTransformation. The timestamp is used to log the time when a transformation has occurred and the name logs the name of the corresponding operation.

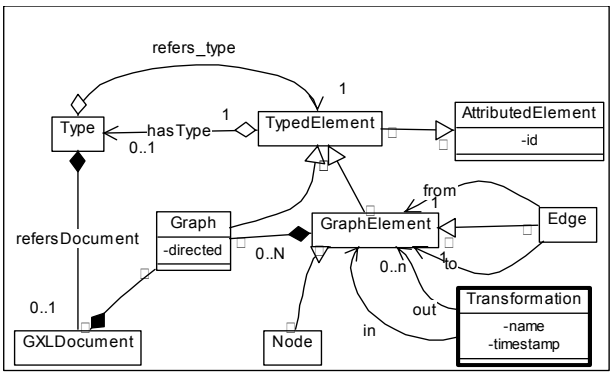

Figure 9. Graph model for History Graphs

The algorithm in Figure 10 describes the work performed by our History Management component. At the beginning, it receives the start graph from the reengineering tool. This start graph (SG) becomes the initial history graph (HG). Then, the History Management component logs all invoked transformations in the history graph. If process iterations occur the History Management component recovers the initial start graph and sends it to the tool. Line 8 specifies that the start graph can be recovered by taking all graph elements from HG that do not represent transformations and do not comprise incoming *out*-edges.

Once the reengineer has completed the modifications to the initial start graph, it is sent back to the History Management component, which computes the changed graph elements (lines 9 and 10). Subsequently, all transformations affected by the changes are determined by transitively traversing *in*- and *out*-edges in the History Graph. Note that we use regular path expressions to represent graph traversals, as proposed in [19]. Finally, all affected transformations are removed from the History Graph (Line 12) and the updated tool graph is computed and sent to the tool (lines 13 and 14). This is illustrated in Figure 11 where the star represents a changed graph element in the start graph and the circle

marks dependent graph elements that are to be removed from the History Graph. In this figure, the bold G-nodes represent the updated tool graph that is sent back to the tool.

```
Algorithm History Management
Begin

1. Receive Start Graph from tool (addGraph(SG))
2. HG := SG; // SG becomes the initial History Graph

3. Repeat

4.  Repeat
5.   Log tool transformation in HG (addTransformation)
6.  Until Start Graph requested (getGraph)

7.  Recover Start Graph and send it to tool
8.       SG := {e∈HG | type(e)≠"Trafo" ∧ e.(<-Out-)=∅}

9.  Receive changed Start Graph (addChanges(c))

10. Receive progagate command (propagateChanges)
    // determine all affected transformations
11. AffectedTrafos:= c.(-in->).(-out->&-in->)*

    // undo affected transformations
12. HG:= HG-(AffectedTrafos.-out-> ∪ AffectedTrafos)

13. Compute updated tool graph and send to tool
14.      TG := {e∈HG | type(e)≠"Trafo" ∧ e.(-In->)=∅}

15. Display Undo Report (AffectedTrafos)

16.Loop
End.
```

Figure 10. Algorithm History Management

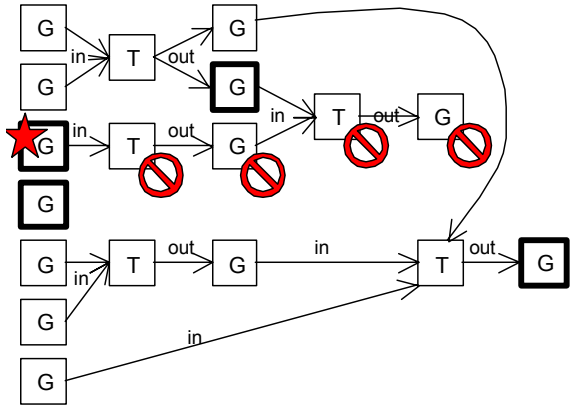

Figure 11. Updated and pruned History Graph

5. Application example: Design Pattern Recovery

The first application of our History Mechanism has been in the domain of database design recovery. We reported on this case study in a previous paper [9]. Since then, we have worked on a generalization of our approach to extend existing tools in various other domains. In the following, we describe an application example in the domain of design pattern recovery. The notion of design patterns has been introduced by Gamma et al. as a way of communicating design solutions for frequently recurring problems [20]. Using design patterns adds a level of abstraction that facilitates description and understanding of software systems. Recently, researchers have begun to use the notion of patterns in the design recovery and documentation of legacy software systems, e.g., [21].

In our study, we have used *UMLStudio*[6], a commercial UML design tool, for recovering design patterns from the *Abstract Window Toolkit* (AWT)[7]. We have chosen UMLStudio because it has advanced mechanisms for end-user programmability: UMLStudio provides a scripting language called *PragScript*. PragScript is a LISP dialect and facilitates traversal of the entire internal data structure of the tool. UMLStudio is shipped with a collection of sample scripts and complete online documentation. It took us approximately four days of work to develop the required GXL export/import functionality by "customizing" these scripts.

UMLStudio provides (limited) functionality for reverse engineering UML class diagrams from Java source code. However, it does not provide automated means for detecting the existence of design patterns. They have to be detected interactively by adding pattern annotations to the constituting design elements. For this purpose, we added a *pattern annotation function* to UMLStudio. The screen shot in Figure 12 illustrates this approach for an excerpt of six sample classes of the AWT[8]. When the reengineer has discovered an instance of a design pattern, (s)he can select its constituents and invoke the pattern annotation function. In our screenshot, the reengineer has recovered three instances of design patterns, namely *Association*, *Composite*, and *Delegation*. Some design patterns depend on the existence of others, e.g., the *Composite* depends on *Delegation* and *Association*. We were unable to visualize the direction of such dependencies in UMLStudio. Nevertheless, we believe that this does not cause a real problem for users who know the design patterns they are trying to detect.

[6] www.pragsoft.com
[7] http://java.sun.com/products/jdk/awt
[8] For simplicity reasons, we compressed the diagram by hiding operations and attributes that are not required for the discussion of our example.

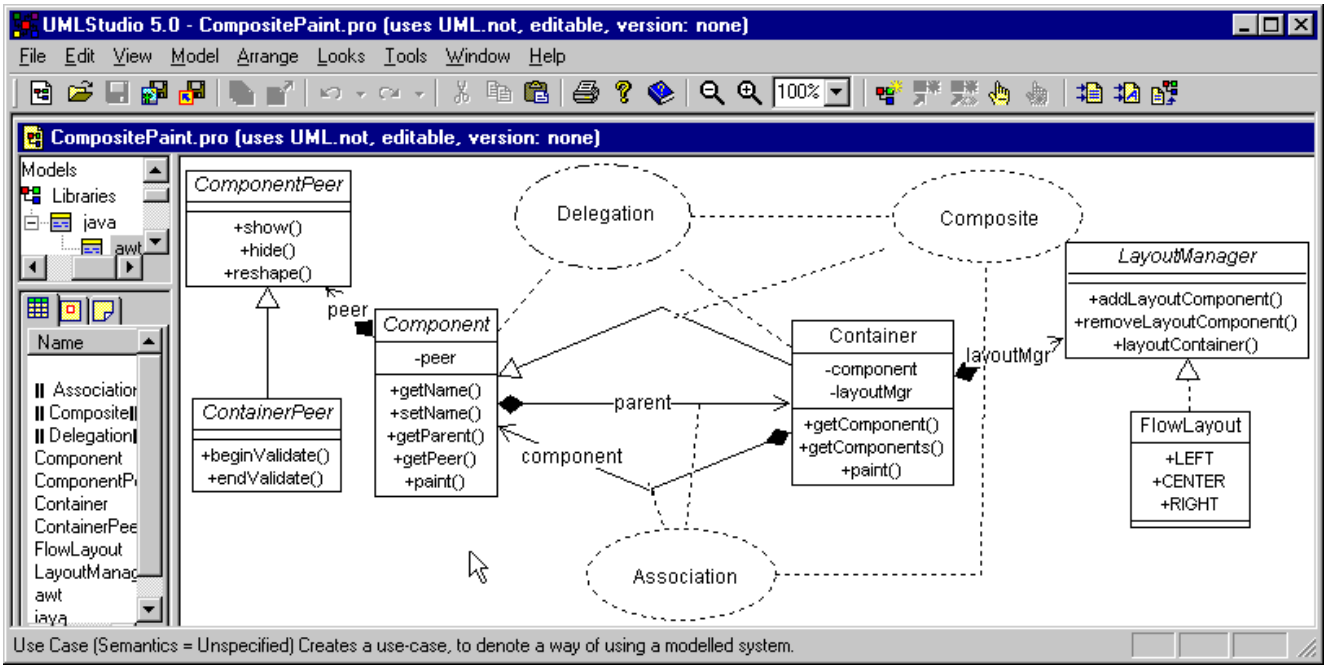

Figure 12. Design pattern detection with UMLStudio

In our theoretical model, we can treat the implemented pattern annotation function as a graph production. If we use our History concept to log transformations necessary for recovering and annotating the three design patterns in Figure 12, we obtain the History Graph in Figure 13: the first annotation transformation makes the information explicit that the two references *parent* and *component* between *Component* and *Container* constitute a single association. The second transformation annotates the existence of a delegation between the two *paint* methods in both classes. Of course, the recovery of this pattern requires additional knowledge, which indicates that a delegation indeed happens between the two operations. One possibility to retrieve this information is to investigate the implementation of these operations. Another frequently used heuristic takes equally named operations as an indication for a delegation. This heuristic was used in our example in Figure 12. Finally, the third design pattern (*Composite*) is based on the existence of the delegation, an association and an inheritance relationship.

Let us assume that the reengineer discovered at a later point that the naming heuristic did not apply in this case and the name equality between both operations is just coincident. Then, the *Delegation* annotation (and all other dependent transformations) have to be undone to avoid inconsistency. Using the History component, the reengineer can go back to the initial version of the class model and enter the information that the knowledge about the *paint* operations has changed. Then, the propagation algorithm would automatically undo all dependent operations while the other manual recovery work is consistently preserved.

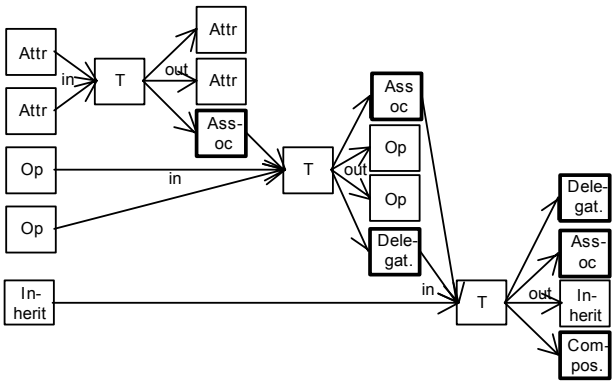

Figure 13. History Graph for design pattern detection

6. Conclusions and future work

Mechanisms that allow for iterative and explorative refactoring processes in human-centered reengineering tools are crucial for the usability of these tools. Only few currently existing tools provide such mechanisms [9, 18]. Moreover, these mechanisms are usually tightly integrated with the rest of the tool and cannot be reused in other tool environments. The approach presented in this paper is based on our practical experiences with one of these mechanisms [9] and attempts to generalize the involved concepts to make them reusable. The recent

movement of the reengineering community towards tool-interoperability based on a common graph exchange format (GXL) opens up an opportunity to develop reusable "backbone" components like the History Manager described in this paper.

In a previous paper, we have shown that our approach is feasible and scalable [9]. This application was in the domain of data reverse engineering. This paper demonstrates that the History concept can be generalized for other application domains (recovery of object-oriented design patterns) and integrated with existing reengineering tools. Our future work will be on improving the interfacability and usability of our History component. We intend to implement a user interface with a history browser.

Acknowledgements

The described research has been supported in part by the National Science and Research Counsel of Canada (NSERC). Many thanks to Vladislav Krasnyanskiy for his support in implementing the current History Management component.

References

[1] Muller, H., S. Tilley, and K. Wong. *Understanding Software Systems using Reverse Engineering Technology: Perspectives from the Rigi Project*. in *CASCON '93*. 1993. Toronto Ontario, Canada: IBM.

[2] Storey, M.A. and H. Muller. *Manipulating and Documenting Software Structures using SHriMP Views*. in *International Conference in Software Maintenance*. 1995: IEEE CS Press.

[3] Henrard, J., et al. *Program Understanding in Database Reverse Engineering*. in *DEXA'98*. 1998. Vienna, Austria.

[4] Ebert, J., B. Kullbach, and A. Panse, *The Extract-Transform-Rewrite Cycle - A Step towards MetaCARE*. 1997, Fachberichte Informatik, Universität Koblenz-Landau: Koblenz.

[5] Cremer, K. *A Tool Supporting the Re-Design of Legacy Applications*. in *2nd Euromicro Conference on Software Maintenance and Reengineering*. 1998: IEEE-CS Press.

[6] Grisworld, W.G., et al., *Tool Support for Planning the Restructuring of Data Abstractions in Large Systems.* IEEE Transactions on Software Engineering, 1998. **24**(7): p. 534-558.

[7] Jahnke, J.H. and A. Zundorf. *Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment*. in *Theory and Application of Graph Transformations*. 1998. Paderborn, Germany: University of Paderborn, Germany.

[8] Jahnke, J.H. and A. Walenstein. *Reverse Engineering Tools as Media for Imperfect Knowledge*. in *Working Conference on Reverse Engineering (WCRE 2000)*. 2000. Brisbane, Australia: IEEE Computer Society.

[9] Jahnke, J.H. and J.P. Wadsack. *Integration of analysis and redesign activities in information system reengineering*. in *3rd European Conference on Software Maintenance and Reengineering (CSMR'99)*. 1999. Amsterdam, NL: IEEE Computer Society.

[10] Rozenberg, G., ed. *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 1. 1999, World Scientific.

[11] Corradini, A., U. Montanari, and F. Rossi, *Graph processes.* Fundamenta Informaticae, IOS Press, 1996. **26**(3): p. 241-265.

[12] Holt, R., et al. *GXL: Towards a Standard Exchange Format*. in *Working Conference on Reverse Engineering (WCRE)*. 2000. Brisbane, Australia: IEEE-CS Press.

[13] Simpson, M., *CVS Version Control and Branch Management.* Dr. Dobb's Journal of Software Tools, 2000. **10**(25): p. 108,110-114.

[14] Lefering, C. and A. Schurr, *Specification of Integration Tools*, in *Building tightly integrated software development environments - The IPSEN Project*, M. Nagl, Editor. 1996, Springer Verlag: Berlin.

[15] Nagl, M., ed. *Building tightly integrated software development environments - The IPSEN Project*. LNCS. Vol. 1170. 1996, Springer: Berlin.

[16] Rajala, N., D. Campara, and N. Mansurov. *Insight - Reverse Engineer Case Tool*. in *Intl. Software Engineering Conference (ICSE-'99)*. 1999: ACM Press.

[17] Kullbach, B., et al. *Program Comprehension in Multi-Language Systems*. in *Working Conference on Reverse Engineering (WCRE)*. 1998. Hawaii, USA: IEEE-CS Press.

[18] Hick, J.-M., et al. *Strategies pour l'evolution des applications de bases de donnees relationnelles : l'approche DB-MAIN*. in *XVIIe congress INFORSID, La Garde, France.* 1999.

[19] Schurr, A., A. Winter, and A. Zundorf. *Graph Grammar Engineering with PROGRES*. in *European Software Engineering Conference (ESEC)*. 1995: Springer Verlag.

[20] Gamma, E., et al., *Design patterns : elements of reusable object-oriented software*. Addison-Wesley professional computing series. 1995, Reading, Mass.: Addison-Wesley. xv, 395.

[21] Keller, R.K., et al., eds. *The SPOOL Approach to Pattern-Based Recovery of Design Components*. Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation, ed. E. H. and O. Tanir. 2001, Springer-Verlag.