Handling Large Search Space in Pattern-based Reverse Engineering[*]

Jörg Niere, Jörg P. Wadsack, Lothar Wendehals
Software Engineering Group
Department of Computer Science, University of Paderborn
Warburger Straße 100, 33098 Paderborn, Germany
[nierej|maroc|lowende]@upb.de

Abstract

Large industrial legacy systems are challenges of reverse-engineering activities. Reverse-engineering approaches use text-search tools based on regular expressions or work on graph representations of programs, such as abstract syntax graphs. Analyzing large legacy systems often fail because of the large search space. Our approach to handle large search space in pattern-based reverse engineering is to allow imprecise results in means of false positives. We use the theory of fuzzy sets to express impreciseness and present our approach on the example of recovering associations.

1. Introduction

Industrial relevant systems often consist of several million lines of code, written in different languages. The variety of the used languages makes reverse engineering of those legacy systems hard to perform. Even in cases where the system has been implemented in only one language, the huge size of the system prevents a complete reverse engineering task.

Reverse engineering is often done using simple text-search tools such as *grep*, *sed* or *awk*. Especially in design-recovery regular expression based tools has been turned out to be inappropriate, because regular expressions are not sufficient to cover a large variety of implementation variants. More sophisticated approaches, e.g. control flow or data flow analyses, require compiler techniques, where the source code is usually represented as an enriched abstract syntax graph.

For analyzing and manipulating graphs, graph-grammars [13] are well-suited. To use graph-grammars for tool-supported design-recovery, so called graph-rewrite-systems apply graph-rewrite-rules on a host-graph (e.g. the abstract syntax tree representation of source code). Unfortunately, applying a graph-rewrite-rule means to solve the subgraph isomorphism problem, which is NP-complete, cf. [9]. Therefore, most approaches are only able to handle some thousand lines of code, cf. [17]. A solution is to reduce the number of rules and/or reduce the host-graph by omitting information, cf. [8]. The first case does not solve the problem because actual legacy systems consist of million lines of code and also split parts consist of several hundred thousand lines of source code. The second case does not solve the problem either because the results are mostly unreliable and contain many false-positives. In addition, filtering the results has to be done manually.

To illustrate our approach, we use a simple design-recovery example, which means recovering associations from Java source code. Recovering associations is a need for further design-recovery activities and therefore an ideal analysis starting point. Since Java is an object-oriented language, it explicitly consists of language constructs for classes, attributes and methods, which correspond directly to classes, attributes and methods in the class diagram. Therefore, parsing the Java source code is sufficient to extract such information.

To illustrate our approach we choose the recovery of associations in Java code. Java has no explicit language construct for associations. Thus associations could be implemented in many different ways. We assume that associations are bidirectional and such composed of two references. Therefore, Figure 1 shows 3 variants of reference implementations.

```
public class House {
  // variant 1:
        public Elevator myElevator;
  // variant 2:
        private Level curLevel;
        public Level getCurLevel() {
          return curLevel; }
        public void setLevel(Level l) {
          if (curLevel != l)
            curLevel = l; }
  // variant 3:
        private TreeSet levels;
        public void addToLevels(Level l) {
          if (!levels.contains (l)) {
            levels.add (l);
            l.setHouse (this); }
        }
        public Iterator iterLevels() {
          return levels.iter(); }
}
```

Figure 1: Reference source code variants

The first two variants of Figure 1 show so called *ToOne*-references, which refer to the multiplicity of the references, i.e. an object instance of one class could have only one related object instance of the other class during runtime. Another multiplicity is *ToMany* (cf. variant 3 in Figure 1). Additionally consider qualified, sorted and n-ary references.

Variant 1 of the reference implementations in Figure 1 is just a public reference to the associated class, whereas variant 2 implements a private attribute of the type of the

[*]This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-1.

associated class including public read/write access methods. These two variants indicate that they are implementations of references from class House to class Elevator and class Level, respectively.

Variant 3 differs from the first two by using a container class from the Java Foundation Classes (JFC), namely a TreeSet. The use of a container class indicates that this is a *ToMany*-reference and the container type indicates that this is a 'normal' reference, e.g. qualified references use container classes which store elements in hash tables for direct access.

The kind of objects that are stored in the container can only be derived by analyzing the addToLevels access method. The parameter is a Level and this object is stored in the container by the add method of the container. Therefore, the implementation is a *ToMany*-reference between class House and class Level.

In general, the flexibility provided by programming languages leads to the problem that different programmers use different syntax to describe the same semantics. However, on design-level only the semantics is relevant. Therefore, each implementation variant has to be defined, in order to make them detectable by a tool. Consequently, using graph-rewrite-rules, each variant is represented by at least one rule, which means a large number of rules precisely describing exact one implementation variant. The large number of rules in addition to the large host-graph intensifies the performance problem of design-recovery in a large search space. Reducing the large host-graph or the rules does not solve the problem as stated before.

A possibility is to identify common parts in a number of rules and exchange those rules each precisely describing one variant by at least one common rule describing a number of variants. Using classical graph-rewrite-systems to apply such imprecise rules lead to many false positives and the reengineer is not able to distinguish between false positives and correct results. Our approach is to allow reducing the number but to express the degree of impreciseness by assigning fuzzy beliefs to rules. In combination with filtering the results with fuzzy values higher than a certain threshold, this approach allows a reengineer to profit from a thorough analysis of the source code in an appropriate time-limit and valuing about the found matches for a rule.

2. Related Work

Since recovering associations is a part of design recovery, the related work presented mostly describes design recovery approaches. Recovering associations is thereby included in all approaches.

Harandi and Ning [4] present program analysis based on an Event and Plan Base. The analysis process starts by firing rudimentary events constructed from source code. Plans define the correlation between one or more (incoming) events and they fire a new event corresponding to the intention of the plan. Each plan definition corresponds to exactly one implementation variant, which leads to a high number of definitions. This applies also to the approach of Paul and Prakash [11], where a matching algorithm for syntactic patterns based on a non-deterministic finite automaton is introduced.

An approach to recognize clichés, i.e. commonly used computational structures, is presented by Wills [17]. The approach is part of the GRASPR system, which examines legacy code. Wills uses attributed flow graph grammars, which combine control-flow and data-flow aspects in one graph-rewrite-rule. Since the sub-graph isomorphism problem is NP-complete [9] this approach allows analyzing only some thousand lines of code, which was sufficient to detect data-structures or search and sorting algorithms. Applying the approach to larger programs had failed.

Krämer and Prechelt [8] use PROLOG in order to detect design patterns [3] in C++ source code. The source code is parsed into facts and rules describing the relations. PROLOG's execution mechanism applies the rules in arbitrary sequence and uses back-tracking where necessary. The approach is able to analyze larger programs, but its preciseness is very low, because the approach uses information of header files only. An analysis of method bodies is not supported. An approach producing more precise results is presented by Antoniol et al. [1, 14]. They use metrics, such as the number of method calls within a method body, to include a method body analysis without time-intensive graph-rewrite-rules. Unfortunately, the used metrics are inadequate to express detailed information, e.g. method calls within loops. Therefore a lot of false positives remain.

Analyzing behavior as well as structure using patterns is presented by Keller et al. [7]. They use a common abstract syntax graph model for UML to represent the source code as well as the patterns. Matching the patterns on the program's syntax graph is done by scripts. These scripts are not generated automatically out of the pattern definitions but have to be implemented by the reverse engineer manually. Such scripts very quickly become large, awkward to read and difficult to maintain. Therefore, scripts used in a certain reverse engineering task are usually not sufficient for other reverse engineering processes.

None of the above approaches allow expressing impreciseness. Jahnke [5] uses possibilistic logic to handle uncertainty in a database reverse engineering process. Graph-rewrite-rules applied by the PROGRES system, developed at Aachen University, extract clichés from SQL statements stored in the database. The following inference process combines the extracted clichés according to a specified Fuzzy Reasoning Net. We [6] used this approach to detect possible memory leaks in applications for Java Smart Cards. Jahnke's approach was successful for the database reverse engineering but the approach has failed for the detection of memory leaks. This results from the fact, that only a small number of rules are needed to extract cliché variants from SQL statements but a huge number of rules is needed to extract implementation variants from Java Smart Card source code.

3. Rule Definition

In our approach to pattern-based design-recovery, we define graph-rewrite-rules with respect to the abstract syntax graph (ASG) representation of a program. The rule definition is graph based, means the approach needs only a graph representation of the code or any other graph representing information of a system, e.g. data-flow or control-flow graphs. Therefore the presented approach is not bound to any particular program language or any particular programming paradigm. For our example, we use a simplified ASG model for readability reasons.

Before defining graph-rewrite-rules for implementation variants, Figure 2 shows an excerpt of the type graph model underlying the rules as an UML class diagram. We omit the complete graph model and just present two classes namely Node and Annotation. Each construct in the code (abstract

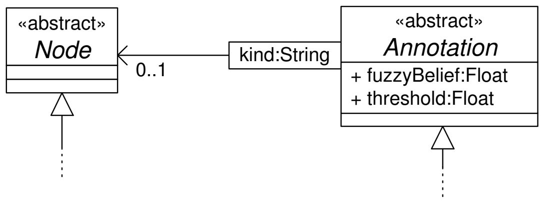

Figure 2: Type Graph Model

syntax graph) is represented by a corresponding subclass of class Node. Each annotation used in a rule is a subclass of class Annotation. The Annotation nodes are associated to the Node nodes in the graph by a qualified association. The qualifying attribute kind is represented in the rules as the name of the corresponding link, e.g. field and references in Figure 3.

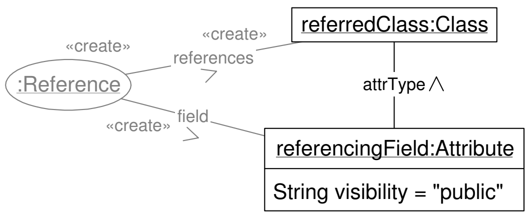

Figure 3: Reference Rule, variant 1

A graph-rewrite-rule is composed of two parts. The pattern part, i.e. the subgraph specifying the application place of the rule in the host-graph and the part describing modifications. Here, only new nodes (annotations) and links are created. In the notation used, the subgraph to be matched in the host graph is defined by the black nodes and edges. The subgraph to be added is defined by the gray node(s) and edges annotated with the stereotype ≪create≫. Note that the Annotation nodes can also be used by other rules. The instances of the Node and Annotation class are represented by rectangle- and oval-shapes respectively.

As a first example of such a rule definition, Figure 3 shows the graph-rewrite-rule defining a public reference between two classes. In the ASG we find a referencingField to a class, i.e. we recognize an attribute that is a public reference to a class. The oval-shaped node is the annotation which identifies the subgraph of the ASG as match of a reference. During rule application this node including the edges are created and thus, enrich the ASG semantically.

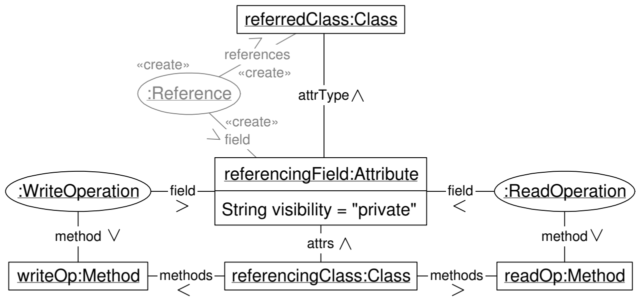

Figure 4: Reference Rule, variant 2

Variant 2 of Figure 1 also shows a reference, the corresponding rule is depicted in Figure 4. In addition to a referencing attribute we require the existence of a read and write operation. These two operations are supposed already to be recovered by other graph-rewrite-rules. Note, that we use the annotations as part of the ASG. This facilitates the definition of our recovery rules by defining rules that are build on the results of other rules.

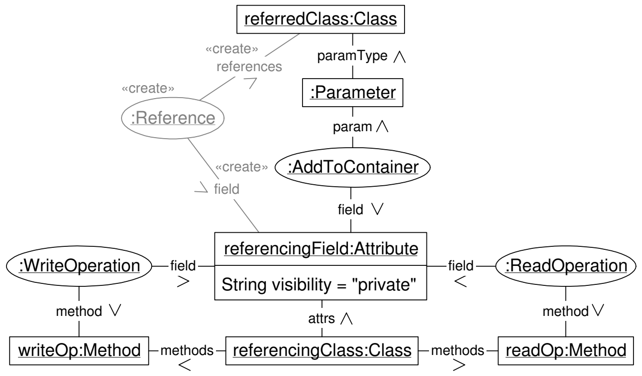

Figure 5: Reference Rule, variant 3

Figure 5 shows the graph-rewrite-rule for variant 3. The higher complexity results from the referencing container instead of the *ToOne*-reference. In order to detect such a *To-Many*-reference we have to look into the method body.

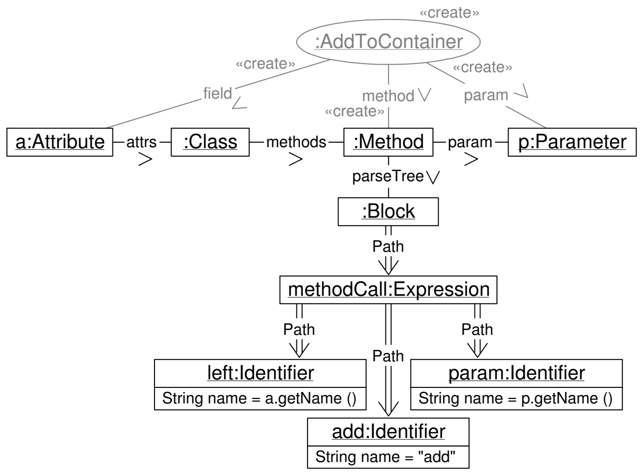

Figure 6: AddToContainer Rule

An AddToContainer requires the existence of an attribute definition in one class, that must be defined as a collection which contains objects of the type of the other class. The definition of the AddToContainer rule is depicted in Figure 6. The method body of that class must contain a method call of an add operation, which is the method for adding elements to the container. Each edge labeled Path in the rule indicates that an arbitrarily path must exist between the source and target node of that edge, i.e. the call can appear in an arbitrarily deep nesting of statements within a method body.

Recognizing an association in the ASG during analysis results in the addition of the corresponding annotation. We

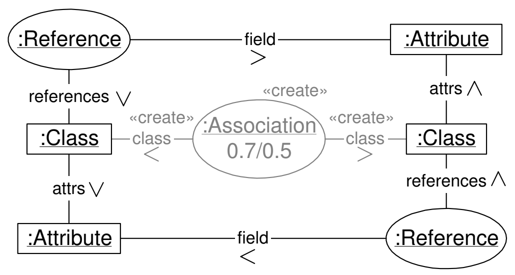

Figure 7: Association Rule

presume that associations are bidirectional implemented, otherwise we call them references. Thus, an association is composed of two references. Figure 7 shows the rule definition. It is sufficient to have a couple of attributes referencing mutually the class they belong to. The association rule is simple because we use the Reference annotations to express the mutual referencing. Note, in cases where multiple references between classes exist, we will detect an association for each possible reference pair. To avoid such ambiguous detections further analysis is required.

4. Managing Impreciseness

An exact graph-rewrite-rule for each implementation variant has the benefit of few false positives in the analysis results, but this leads to performance problems in large search space because of the high number of rules. We introduce further abstraction into the rule definitions so that one or only few rules cover a large variety of implementations. This reduces the number of rules and improves the performance.

To define a new graph-rewrite-rule with further abstraction the reengineer, first, chooses a set of implementation variants that should be covered by the new rule. Common parts of these implementation variants has to be identified. The common parts, then, form the new graph-rewrite-rule. All implementation variants from the set are at least covered by this new rule. In addition, there will be implementation variants that were not intended to be found by the rule. Some of these are not instances of the searched pattern (false positives). Others are correct matches that were not considered when defining the graph-rewrite-rule. This is a kind of impreciseness that has to be managed.

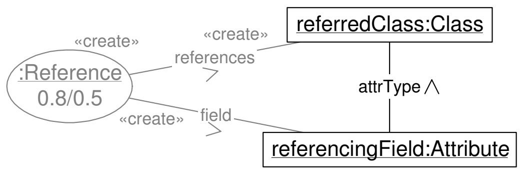

Figure 8: Imprecise Reference rule with fuzzy belief

Figure 8 depicts a new graph-rewrite-rule that is designed to replace all three rules from Figures 3 to 5. It covers all references without any constraints to the visibility. Therefore, it is evident that variant 1 depicted in Figure 1 is covered by the new rule. The only difference between the old rule for variant 1 and the new rule is the missing constraint of the Attribute node.

Variant 2 of Figure 1 - a private reference - is also covered by the new rule, because the visibility of the attribute is ignored by the new rule. The read and write access methods are not required any more.

The new rule finds a different match for variant 3 of Figure 1 than the original rule in Figure 5. A *ToOne*-reference between classes House and TreeSet is found instead of a *ToMany*-reference between House and Level.

The purpose of an attribute of a container type is to implement a *ToMany*-reference to another class. In this case a *ToMany*-reference is implemented between House and Level. This reference could not be detected by the rule in Figure 8. The reference found is a false positive in the way that a reverse engineer would ignore a reference to a container class. The more relevant information is the *ToMany*-reference to class Level. Note, this false positive is not found by the first three precise Reference rule definitions.

The impreciseness of a graph-rewrite-rule that stems from reducing the number of rules has to be valued to be useful to the reengineer. The value should describe the ratio between correct matches and all matches of a certain rule including false positives, i.e. the preciseness of a rule. After all applications of a rule, this ratio can be calculated. At the time of the rule definition the number of false positives produced by the rule can only be estimated.

In our approach we introduce a so called *fuzzy-belief* for each graph-rewrite-rule that expresses its preciseness. The fuzzy-belief of a rule is a value between 0 and 1. By this value, the reengineer expresses his estimation that 30% of all matches are false positives. Thus, 70% of all matches would be correct and the ratio would be 0.7. In Figure 8 the fuzzy belief is defined in the gray annotation node that is created when the rule is applied. It is the first value in the pair of numbers; the second is explained in the next section.

5. Inference

Pattern-based reverse engineering is a deductive analysis problem where rules are repeatedly applied to a graph representation of the source code. Pure deductive analysis algorithms typically apply the rules involved level by level, bottom-up, according to their natural hierarchy, and produce useful results only when analysis is complete. Results from other researchers, such as [17] and [12], suggest that a reverse engineering tool providing pure bottom-up pattern-based analysis cannot scale for larger software systems. Therefore, we have adopted an analysis algorithm which combines a bottom-up strategy and a top-down strategy. The overall analysis finishes when all possible annotations are created. In the following we only sketch a part of the algorithm to show how the fuzzy values of the results are calculated. The entire algorithm is described in detail in [10].

Rules create annotations with a certain fuzzy value. The fuzzy value is calculated during rule application for each match of the graph-rewrite-rule. They are stored in the fuzzyValue attribute of the annotation nodes (cf. Figure 2). The basis for the fuzzy value calculation are the fuzzy beliefs of the graph-rewrite-rules. The fuzzy value of a match is computed as the minimum of the fuzzy values of all annotation nodes occurring in the match and the fuzzy belief of the rule. Thus, calculation of the fuzzy values is similar to fuzzy grammars, cf. [18].

To apply the Association rule of Figure 7 two annotations (both of type Reference) are needed. The fuzzy belief of the Association rule is 0.7. In Figure 9 a cut-out of the graph during an inference is shown. The Reference rule has created

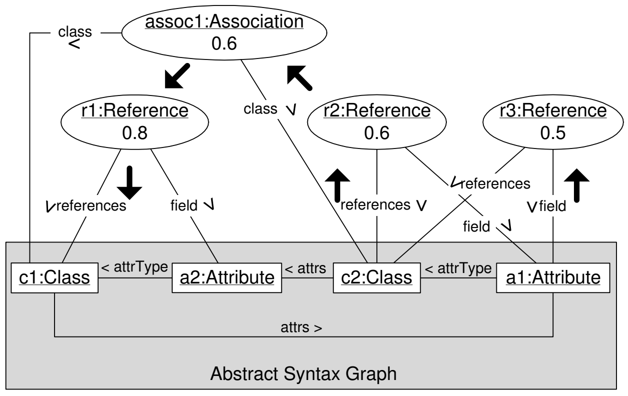

Figure 9: Inference of Association

an annotation r2 with value 0.6 on attribute a1. Upon this r2 annotation the Association rule has created annotation assoc1. Since an association needs two references a top-down analysis is performed to detect the second reference r1 with value 0.8. A fuzzy value of 0.6 is assigned to the assoc1 annotation node, i.e. the minimum of 0.8, 0.6 and 0.7. Later the annotation r3 with value 0.5 is detected. This annotation is not considered furthermore because it has a fuzzy value of 0.5 which is lower than the fuzzy value 0.6 of annotation r2 used in the match of the association.

Assume that the fuzzy values of the Reference annotations are both 0.8, the fuzzy belief of 0.7 of the Association rule would be the minimum. So the fuzzy belief of the graph-rewrite-rule is an upper bound for the fuzzy values computed.

A node in the graph-rewrite-rule may have multiple valid matching annotation nodes in the host graph that have different fuzzy values. In that case the annotation node with the maximum fuzzy value is chosen as match such that the new annotation created by the rule application uses the most reliable source of information. Therefore, the fuzzy value of an annotation corresponds to the maximum fuzzy value of all derivations of the annotation, cf. [18].

As a way to limit the rule applications to reasonable cases we introduced thresholds to graph-rewrite-rules. In Figure 7 the threshold is defined in the gray annotation node that is created when the rule is applied. It is the second value (0.5) in the pair. If any annotation that is part of the subgraph to match has a fuzzy value lower than 0.5, the rule would not be applied. This helps to minimize computation time and memory resources, that would otherwise be used for investigation of unreliable and imprecise information. Thus, the thresholds improve the scalability of our approach. They are chosen by the rule developer based on personal experience and/or historical data.

6. Feasibility Test

This section presents the result of our approach integrated in the FUJABA environment. The graph-rewrite-rules, in FUJABA so called story-patterns, are automatically translated into Java source code, which could be compiled with a conventional Java compiler. The rule dependencies are exported into an XML file, which the inference engine is able to execute. The inference engine itself has been implemented using FUJABA. The test has been performed on a Linux PC, 1.7 GHz, 1GB RAM, and JDK 1.4.1.

In the scenario of this feasibility test we run two analyses on the same source code, i.e. Java's AWT window toolkit library comprising 140.000 LOC in 345 classes. In the first run we have used a catalog with rules describing exact implementation variants of references. The catalog includes 8 rules where some are described in this paper. The second catalog we used in the second run includes only 2 'fuzzy' rules. These are the Reference and Association rules of Figures 8 and 7, respectively.

Catalog	#Rules	#References	#Assocs	Duration
1	8	1122	6	9:12 min.
2	2	1511	5	0:34 min.

Table 1: Results from evaluation

Table 1 shows the results of the test. The first row shows the numbers for the run with the exact rules and the second one the run with the 'fuzzy' rules. According to the above described reduction of rule numbers by identifying common parts in a set of rules and replacing them with at least one other rule, the number of found references in the second run is higher than in the first run. In contrast to this, the number of detected associations in the second run is lower than in the first. An interesting fact is the time spend for each analysis, means an 18 times faster analysis by reducing the catalog by 6 rules, only.

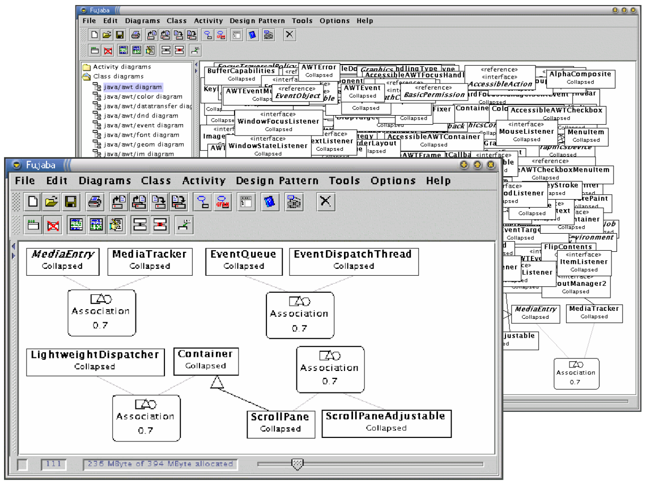

Figure 10: Association recognition in Java AWT

This speed-up, in context of a reference detection, is not very useful, because the reengineer has to filter the false positive references manually. In the context of association detection this speed-up is remarkable, because in both runs all *OneToOne*-associations have been detected. Figure 10 depicts screenshots of FUJABA analyzing the AWT library. The rear screenshot shows the class diagram of the AWT library, the front screenshot shows a view to the class diagram with some of the association annotations detected during the second run. The one association not detected in the second run is the *OneToMany*-association between class Component and class Container.

The reason for not detecting the *OneToMany*-association is the effect, that all rules responsible for the detection of references, even the ones for detecting *ToMany*-references, have been replaced by the rule shown in Figure 8. Therefore, all original *ToMany*-references using containers have been detected as *ToOne*-references to the container, which prevents detecting of *OneToMany*-associations. In addition, the 300 false positive references found are originally attributes, which store data such as sizes and positions of elements on the screen.

7. Conclusions

Reverse engineering based on pattern-based techniques incorporate the problem that they could not be applied on analysis of large systems, because they inherit the NP-complete problem of subgraph isomorphism. Our approach presented in this paper handles large search space in pattern-based reverse-engineering activities by reducing the number of graph-rewrite-rules used for the analysis. We handle the resulting impreciseness, by assigning fuzzy beliefs to each rule, expressing its preciseness.

The example given, recovering associations from Java programs, is a base for further design-recovery or architectural recovery [2] activities and has successfully been applied to recover design patterns in the AWT window library of Java [10]. In addition, we are using this approach to recover Web information systems, where the connection between different data bases are manifested in the source code of applications [15].

However, the question of good choices for fuzzy beliefs still remains. The fuzzy beliefs assigned to rules should express the ratio of the correct matches to all matches of the rule including false positives. Since this question could only be answered during the analysis of a system, we are currently working on an automated adjustment of the fuzzy beliefs during runtime. First results have shown that the adjustment rules are simple, which means only a small overhead but a large improvement of the results.

Our approach is restricted to static analysis, which limits the recognition of dynamic parts of design patterns. Method invocations in object-oriented languages with polymorphism and dynamic method binding can not be analyzed correctly by static analyses. The concrete invoked method and the concrete object the method is invoked on can only be analyzed during runtime. Therefore, dynamic information has to be incorporated in the analysis of a system. We are currently working on an approach [16] to compare UML sequence diagrams to method traces gathered during runtime.

8. References

[1] G. Antoniol, R. Fiutem, and L. Christoforetti. Design pattern recovery in object-oriented software. In *Proc. of the* 6th *International Workshop on Program Comprehension (IWPC), Ischia, Italy*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Somerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* John Wiley and Sons, Inc., 1st edition, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[4] M. T. Harandi and J. Q. Ning. Knowledge based program analysis. *IEEE Transactions on Software Engineering*, 7(1):74–81, 1990.

[5] J. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes.* PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[6] J. Jahnke, J. Niere, and J. Wadsack. Automated quality analysis of component software for embedded systems. In *Proc. of the* 8th *International Workshop on Program Comprehension (IWPC), Limerick, Irland*, pages 18–26. IEEE Computer Society Press, June 2000.

[7] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proc. of the* 21st *International Conference on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[8] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the* 3rd *Working Conference on Reverse Engineering (WCRE), Monterey, CA*, pages 208–215. IEEE Computer Society Press, November 1996.

[9] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* Springer Verlag, 1st edition, 1984.

[10] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24th *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348, May 2002.

[11] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[12] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[13] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[14] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proc. of the* 9th *International Conference on Software Maintenance (ICSM), Oxford, UK.*, pages 230–238. IEEE Computer Society Press, September 1999.

[15] J. Wadsack, J. Niere, H. Giese, and J. Jahnke. Towards data dependency detection in web information systems. In *Proc. of the Database Maintenance and Reengineering Workshop (DBMR'2002), Montral, Canada. (ICSM 2002 Workshop)*, October 2002.

[16] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003. (to appear).

[17] L. Wills. Using attributed flow graph parsing to recognize programs. In *Proc. of International Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

[18] L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.