An Interactive and Scalable Approach to Design Pattern Recovery[*]

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

Albert Zündorf
Technical University of Braunschweig
Institute of Software
Postfach 3329
38023 Braunschweig, Germany
zuendorf@ips.cs.tu-bs.de

ABSTRACT

Reverse engineering is a process highly influenced by assumptions and hypotheses of a reverse engineer, who has to analyse a system manually, because tools are often not applicable to large systems with many different implementation styles. Successful tools have to support an interactive process, where the engineer is able to steer the analysis process by proving certain assumptions and hypotheses. Consequently, the input format of the analysis tool must support a kind of impreciseness to formulate weak presumptions. In this paper we present a reverse engineering process based on fuzzy graph transformation rules. We use graph rewrite rules in addition with fuzzy logic to detect design patterns in Java source code. Impreciseness is expressed by assigning fuzzy values to graph transformation rules and thresholds are used to look up only firmed occurrences of patterns. Underlying the transformation rules is an object-oriented graph model providing composition and inheritance, which reduces the complexity of the rules. We propose a reverse engineering process starting with imprecise rules and refining and specifying the rules during the analysis. Preliminary results applying our process are promising, i.e., we present the results of detecting design patterns in Java's Abstract Window Toolkit (AWT) library.

1. INTRODUCTION

Reverse engineering tries to recover design information from legacy source code. Simple reverse engineering approaches, using tools like *grep* and *perl*, have turned out as inappropriate for most of the interesting reverse engineering tasks, cf. [6]. More sophisticated results require the analysis of method bodies using compiler techniques. In such sophisticated approaches, the program is usually represented as an enriched abstract syntax graph, cf. [8]. For the analysis of such abstract syntax graphs, graph grammars are well suited, cf. [12, 4]. However, all these approaches face severe scalability and pattern complexity problems.

First of all, large system sizes of several million lines of code need to be managed. Second, the flexibility provided by programming languages give different programmers a lot of freedom to implement a certain pattern in various ways. To deal with all these implementation variants, a large number of variant design element detection rules are required. This large number of rules again leads to a performance and complexity problem.

To overcome these problems, our approach provides a bundle of techniques. We use sophisticated query optimization techniques for the execution of graph rewrite rules, cf. [10, 1]. We allow super-classes as wildcards for sets of node labels within graph rewrite rules. A sophisticated rule selection algorithm optimizes the analysis response time by focusing on the local context of certain pattern indicators. And, instead of a large number of 100% precise descriptions of each possible implementation variant for a certain pattern, we use a small number of somewhat imprecise detection rules, cf. [9].

Our imprecise detection rules may, e.g. only check for some important implementation fragments that are in common to many implementation variants. This reduces the number of necessary pattern definition rules significantly. However, this advantage is paid by a loss of preciseness. Depending on the actual implementation variants used in the considered legacy system, this loss of preciseness may result in false positives, incorrectly found pattern occurrences. In other cases, certain implementation variants may not be covered. This impreciseness depends on individual programming styles and skills of the legacy system developers as well as on the application domain and the programming language used. To deal with this impreciseness in a more flexible way, this paper adds the usage of fuzzy techniques to the pattern definitions presented in [9]. Each detection rule is equipped with a so-called fuzzy belief. This value expresses the confidence of the re-engineer in the rule within the current application context. Furthermore, each rule has a threshold, that restricts the rule application to cases with reasonable confidence. If a pattern definition rule relies on intermediate results of other pattern definition rules, the fuzzy belief of the generated re-

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

sults is computed from the fuzzy belief attached to the rule combined with the fuzzy beliefs of the intermediate results.

The next section 2 presents the related work concerning other pattern detection approaches. Section 3 introduces our fuzzy pattern definitions. Section 4 discusses an sample catalog of detection rules for the "Gang of Four" design patterns [2]. Section 5 gives a brief introduction about our reverse engineering process. The re-engineer's interaction with the process and some results from re-engineering a real world example are topics of section 6. Section 7 finally summarizes our results and shows the direction of our current work.

2. RELATED WORK

Comparable work on reverse engineering of source code has been reported over the past decade. Harandi and Ning [3] present program analysis based on an Event Base and a Plan Base. Events are constructed from source code and plans are used to define the correlation between one or more (incoming) events and they fire a new event which corresponds to the intention of the plan. Each plan definition corresponds to exactly one implementation variant, which leads to a high number of definitions.

An approach to recognize clichés, i.e. commonly used computational structures, is presented in [12]. The GRASPR system examines legacy code represented as flow graphs and clichés are encoded as an attributed graph grammar. The recognition of clichés corresponds to the sub-graph isomorphism problem, which is NP-complete [7]. Consequently, the approach is limited to systems of a few thousand lines of code. Real world systems usually exceed this size.

Keller et al. [5] analyze behavior as well as structure and use the CDIF format for UML to represent the source code as well as the patterns. Scripts match the pattern's syntax graphs on the program's syntax graph. The reverse engineer has to implement the scripts manually, thus they are not generated out of the patterns themselves. Such a script language very quickly becomes large, awkward to read and difficult to maintain and reuse.

In addition, none of the approaches facilitates the exploitation of the re-engineer's domain and context knowledge. This contributes to scalability problems for the process enabled.

3. FUZZY PATTERN DEFINITION

To model fuzzy pattern definitions, we adopt the FUJABA graph model approach [1, 13]. This approach has been formalized using a set-theory approach in [13]. The legacy system to be analyzed is transformed by the FUJABA environment from source code into an abstract syntax graph (ASG) representation. This graph can be manipulated by graph rewrite rules showing the left- and right-hand side as a single graph in a UML collaboration diagram like notation.

For the purpose of pattern definitions, we have adapted the FUJABA notation of defining graph rewrite rules by introducing special shapes for design patterns artifacts and by adding fuzzy beliefs to graph rewrite rules, cf. figures 1 and 3. Those pattern definitions can be applied as graph rewrite rules to the ASG. Pattern definitions are special graph rewrite rules in the way that they don't remove elements from the ASG. Only annotation elements indicating a found pattern occurrence are added to the ASG. Each pattern definition rule adds a single annotation element connected by links to ASG elements. Figure 1 depicts a pattern definition for the "Gang of Four" (GoF) design pattern *Bridge* from [2]. Elements with the stereotype ≪create≫ only belong to the right-hand side of the rule, all other elements belong to both sides. Annotation elements added to the ASG are depicted as oval shaped nodes. Thus already defined patterns can be combined to new pattern definitions.

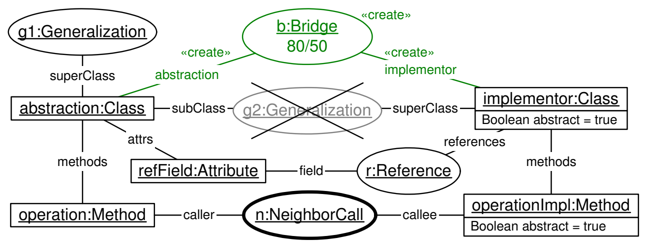

Figure 1: Bridge Pattern Rule

One important feature of the FUJABA graph model is the support for object-oriented inheritance. In pattern definitions super-classes may be used as wildcard node labels that match nodes of all direct and indirect sub-classes. This mechanism has already been proposed in PROGRES, cf. [10]. For example, the Reference node r in bridge may map on a host graph node with label Reference, MultiReference or ArrayReference, cf. figure 2. Similarly, the NeighborCall node n matches NeighborCall and Delegation nodes and the Generalization nodes match Generalization or MultiLevelGeneralization nodes. Thus, the usage of inheritance allows us to replace 3*2*2*2=24 rules with special node labels by a single rule with the super-class node labels.

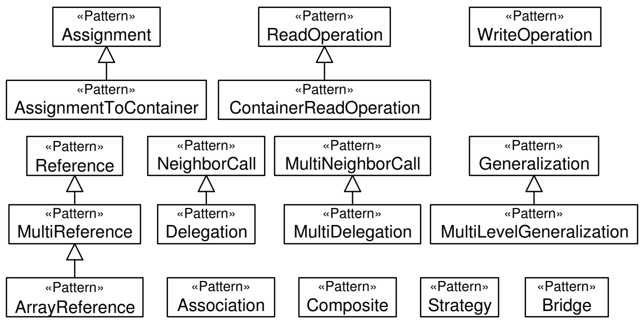

Figure 2: The pattern catalog

Constraints on attribute values may be shown as boolean expressions within the "attribute compartment" of a node. Crossed out nodes or edges represent negative application conditions. For example the crossed out Generalization node g2 in figure 1 represents the negative application condition that the abstraction and implementor classes must not be connected by a Generalization relationship. Note, our rules show some oval nodes with thick border. Such nodes are so-called trigger nodes. They will be discussed in chapter 4.

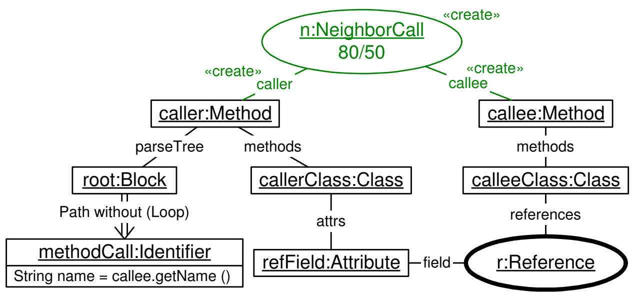

Figure 3: Neighborcall Pattern Rule

Figure 3 depicts a pattern definition for the sub pattern Neighborcall used in the Bridge-pattern definition. It shows how to analyze dynamic parts of patterns. The NeighborCall sub pattern describes a single method call between methods from different classes, where the caller class references the callee class. To detect such a pattern, method bodies have to be analyzed. Implementations of those method calls vary highly. To overcome this problem, only significant parts of the method bodies' ASG are described by the pattern definition. We use paths within the ASG to find these significant elements. In Figure 3 for example an Identifier with the callee method's name is searched within the caller method body. Additionally, the Identifier must not be contained in a Loop, so that this is only a single method call.

For the purpose of fuzzy pattern definition rules, this paper extends the Fujaba rule notation with an implicit handling of fuzzy beliefs. Each oval design pattern annotation node is equipped with two implicit attributes carrying a belief value and a threshold value between 0% and 100%. For example, the "80/50" inscription of the Bridge node in figure 1 gives a fuzzy confidence of 80% and a threshold of 50% for that node/rule. The fuzzy confidence roughly describes the percentage of rule applications that due to the estimation of the rule developer or to the experience reported by historical data have been successfully applied. The remaining percentage thus refer to false positives. In our example we assume that 80 of 100 rule applications actual detect a Bridge pattern while 20 of 100 rule applications mark false positives.

During rule application, the belief of an annotation node is computed as the minimum of the fuzzy beliefs of all annotation nodes employed in that rule. Let's consider we apply the rule of figure 1 and nodes g1, r and n are rated with fuzzy beliefs 60%, 70% and 65%. These values are combined with the 80% fuzzy belief of the Bridge node to be created and the resulting fuzzy belief will be 60%, the minimum of all these fuzzy beliefs. If the fuzzy beliefs of the matches of g1, r and n are all higher than 80%, the belief of the Bridge node would serve as an upper bound for the whole rule: the overall belief can not top 80%.

Note, a node in the reverse engineering rule may have multiple valid matching candidates in the host graph that have different fuzzy beliefs. In that case the candidate with the maximal fuzzy belief is chosen as match such that the new annotation created by the rule application uses the most reliable source of information.

Let us suppose that the fuzzy belief of one of the nodes that are matched by g1, r and n is very low, e.g. 30%. In that case the fuzzy threshold of 50% of the Bridge pattern applies. As mentioned the second fuzzy value employed in our pattern definition rules serves as a threshold that limits rule application to reasonable cases. If some of the intermediate results are very unreliable, i.e. a low fuzzy belief, it may not make sense to waste more computation time and memory resources on further investigations relying on this uncertain information. Thus, thresholds are again a means to improve the scalability of our approach. These thresholds are chosen by the rule developer based on personal experience or based on historical data.

Note, during the inference process new design pattern annotation nodes may be created which have a higher fuzzy belief as already existing similar annotation nodes. This may trigger the re-evaluation of some other rules that formerly used the annotation node with the lower fuzzy belief. Thereby, for some rules the threshold will be met and these rules again create new annotation nodes that may trigger further rule evaluations. However, the fuzzy belief of an annotation may never decrease: if a new node is created with a lower belief than a similar existing one, nothing happens.

4. A CATALOG FOR DESIGN PATTERNS

As stated in the introduction, reverse engineering is naturally an interactive process. Fully automated approaches usually fail, because the large variety of designs and the high number of different implementation styles can not all be recovered by tools in an appropriate time. Current reverse engineering processes support a re-engineer with a number of different tools, e.g. all kinds of *grep* derivatives, which offer plain information extracted from the software system. The re-engineer has to combine the extracted information and make conclusions manually. Other reverse engineering tools with direct focus on design documents such as *Together Control Center* try to detect also relations between classes based on name conventions such as *get-*, *set-* and *add-*, *remove*-prefixes. The produced document is usually a UML class diagram but the techniques to extract the information are comparable with *grep* technology.

Reverse engineering includes, in addition to rudimentary information such as classes and their relations, the recovery of the architecture and behavior of a system and the recovery of dependencies between certain parts of a system. The latter are very important during maintenance of a system, because they show potentially problematic system parts where changes may have many unanticipated side-effects.

Design patterns introduced by Gamma et al. [2] describe good design solutions for recurring problems. Thereby, we use the term design patterns for the certain pattern category and the term GoF-patterns (Gang of Four) for the patterns introduced by Gamma et al. Design patterns describe solutions for more or less complex relations and interactions between different parts in a software system, usually classes in object oriented system designs. For example, a *Bridge* GoF-pattern is a solution to *"Decouple an abstraction from its implementation so that the two can vary independently"* [2, p. 151]. It is often used for window toolkits, and comprises in its application at least 5 classes where each class has to play a certain role.

Consequently, design patterns are highly suited to provide dependencies between parts of a software system. For example, by detecting a *Bridge* GoF-pattern in a system during a reverse engineering task, the dependencies are fixed and this helps to find possible side-effects of changes later on. Currently our reverse engineering approach focuses on the detection of GoF-patterns in Java source code. However, the approach isn't limited neither to GoF-patterns nor to Java. Patterns for different domains are imaginable. The basis for rule application is an abstract syntax graph representation of the source code. So for different languages the parser needs to be changed only.

The acceptance and success of a reverse engineering process does not only depend on its produced results but also on its usability and scalability, especially in semi-automated processes. For semi-automated processes scalability means, in addition to a complete analysis of thousands or million lines of code in an appropriate time-range, to produce reasonable intermediate results quickly. Depending on the intermediate results a engineer is able to steer the analysis process in an early stage and thus avoid non-productive or wrong analysis.

The definition of a GoF-pattern consists of the pattern name, an example application, the static structure, the collaborations, consequences applying the pattern in a systems design and some other parts. All parts are described in prose except the static structure and collaboration, where Gamma et al. used OMT class diagrams resp. collaboration diagrams. This informal definition is not sufficient for a tool supported reverse engineering process. In addition, the informal definition offers many interpretation opportunities, so that patterns are implemented in many different ways even in one application.

As a formal description of design patterns, we use fuzzy graph rewrite rules as described in the previous section. Each pattern is represented by one rule, whereas each rule creates at most one new pattern annotation node with a certain fuzzy belief for each application of the rule.

A cut-out of our patterns and their relations is shown in figure 4. The arcs with a stick arrow-head represent the dependency relations between the pattern and ASG nodes. The pattern catalog contains three GoF-pattern definitions, i.e. the *Bridge*, the *Strategy* and the *Composite* GoF-pattern at the top of the dependency graph in figure 4. All other patterns are used as sub patterns for composition of those GoF-patterns. Additional GoF-patterns could be defined by using the existing sub patterns and possibly defining new sub patterns. The catalog is the result of two analysis processes, the re-engineering of Java's AWT and the Java Generic Library (JGL), and thus adapted to the specific analyzed software systems.

5. THE RE-ENGINEERING PROCESS

Our approach provides a semi-automated iterative reverse engineering process, which is illustrated in figure 5 depicted as a statechart. The rule application, what we call *inference process* is described in detail in [9]. Only a brief introduction of the inference process will be given here.

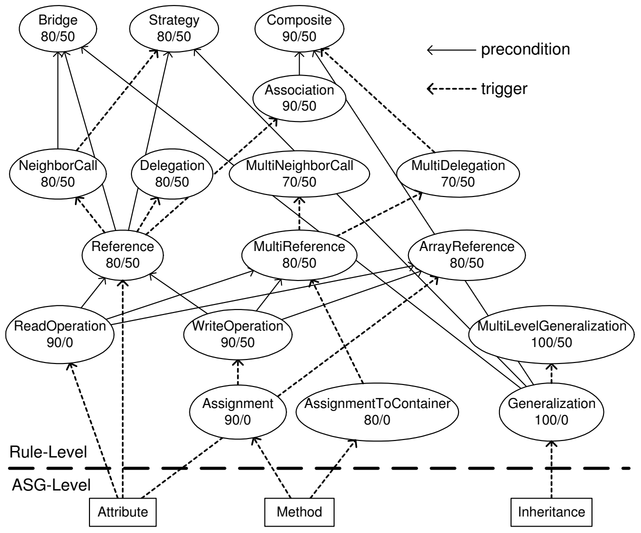

Figure 4: Pattern dependencies

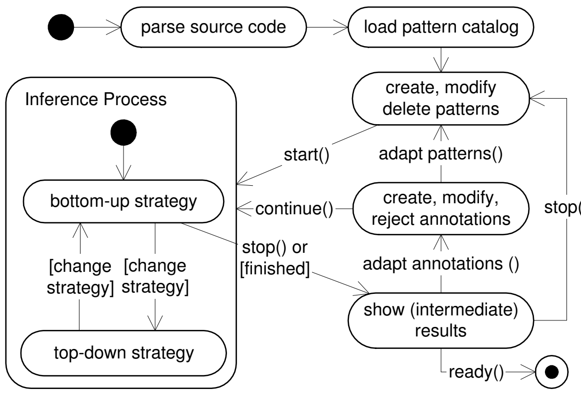

Figure 5: The re-engineering process

The first step in the re-engineering process is parsing the source code and creating an abstract syntax graph representation including any kind of additional links, e.g. application-to-declaration links. In the next step the pattern catalog which seems to be best adapted for the system will be loaded. Afterwards the re-engineer can modify, add or remove rules, which is the start of the iterative process. The inference process starts after all modifications are done.

Figure 4 shows the dependencies between the pattern definitions. This dependency graph is used by the analysis algorithm of the inference process to determine the order of rule applications. The algorithm consists of a combined *bottom-up* and *top-down* search. It starts in *bottom-up* mode at the ASG-level as depicted in figure 6. In this example the application of the Reference rule is tested for a certain element of the ASG, the Attribute a. Before the Reference rule could be applied, two preconditions have to be checked. These are the existence of a ReadOperation and a WriteOperation pattern occurrence. To do so, the algorithm switches

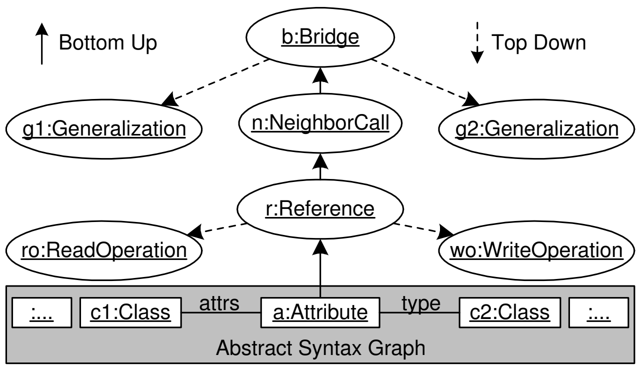

Figure 6: Sample inference process

to the *top-down* mode. Let's consider the rules for these two preconditions could be applied. Then the algorithm switches back into the *bottom-up* mode and applies the Reference rule. An annotation element indicating a found occurrence of the Reference pattern is added to the ASG and linked to the element a. The application of this rule triggers the next rule, namely the NeighborCall, cf. figure 4. The NeighborCall rule triggers again the Bridge rule. For the precondition check the algorithm uses the *top-down* mode.

For each application of a rule one ore more context elements of the preceding rule application are given. This prevents the NP-complete problem of sub graph matching. With context elements the sub graph matching is localized to only a few elements of the whole graph.

After the application of the Bridge rule one step in the iterative inference process is done. No other rule is triggered. The next step of the inference process starts with an element of the ASG again. The inference process ends, if all elements from the ASG are checked for application of certain rules.

6. THE RE-ENGINEER'S INTERACTION

Due to the fact, that this is an iterative process, the re-engineer can interrupt the algorithm each time it is in *bottom-up* mode, cf. figure 5. The analysis resides in an consistent state in the *bottom-up* mode whereas in the *top-down* mode some preconditions of rules may not yet have been checked. The re-engineer is now able to investigate the (intermediate) results and may adapt fuzzy values of certain annotations. Let's consider, the re-engineer agrees with the found Neighborcall pattern occurrence of figure 6. The annotation created by the inference process got a fuzzy belief of 80%. To express, that the found pattern is actually a Neighborcall pattern occurrence, the re-engineer may now raise its fuzzy belief up to 100%. All fuzzy beliefs of consequent annotations have to be recalculated now. This takes usually only a fraction of a second. The fuzzy belief of the Brigde annotation could raise too, if the belief of the Neighborcall pattern occurrence was the limiting value.

On the other hand, the re-engineer may mistrust or even reject the annotation of the Neighborcall pattern. To do so, the fuzzy belief will be decreased to a lower value or even to 0%. Now the threshold of consequent rules could take effect. In this case the fuzzy belief of consequent annotations would drop to 0%.

Furthermore, in addition to the interaction during the analysis, the re-engineer could adapt the pattern definition rules. The adaptation would include the change of the pattern's structure as well as fine tuning by changing the rule's fuzzy belief or threshold. To speed up analysis time, the re-engineer could optimize trigger paths. The trigger path to the Bridge pattern starts with an Attribute within the ASG. This triggers the Reference pattern, which again triggers the Neighborcall pattern, that finally triggers the Bridge pattern. Let's consider another trigger path from the ASG element Inheritance to the Generalization pattern and finally to the Bridge pattern. This trigger path would be worse. It wouldn't change the overall complexity, but a lot more of failing applications of the Bridge rule would arise. So setting up the trigger paths requires to take care.

We have developed a prototype supporting the described reverse engineering process. The prototype is part of the FUJABA environment. It contains editors for the pattern definitions and an inference engine, which uses the pattern matching algorithm provided by FUJABA. We use the JavaCC [11] parser to generate an abstract syntax graph representation of the source code. Intermediate results are shown as enriched UML class diagrams using annotations.

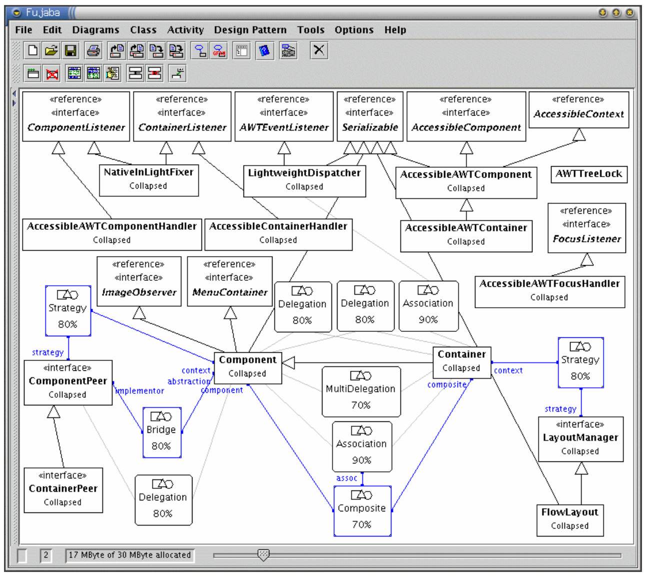
Fujaba
File Edit Diagrams Class Activity Design Pattern Tools Options Help
«reference» «interface» ComponentListener
«reference» «interface» ContainerListener
«reference» «interface» AWTEventListener
«reference» «interface» Serializable
«reference» «interface» AccessibleComponent
«reference» AccessibleContext
NativeInLightFixer Collapsed
LightweightDispatcher Collapsed
AccessibleAWTComponent Collapsed
AWTTreeLock
AccessibleAWTComponentHandler Collapsed
AccessibleContainerHandler Collapsed
AccessibleAWTContainer Collapsed
«reference» «interface» FocusListener
«reference» «interface» ImageObserver
«reference» «interface» MenuContainer
AccessibleAWTFocusHandler Collapsed
Strategy 80%
Delegation 80%
Delegation 80%
Association 90%
strategy
«interface» ComponentPeer Collapsed
Component Collapsed
Container Collapsed
Strategy 80%
context
abstraction
implementor
component
MultiDelegation 70%
composite
context
Bridge 80%
strategy
«interface» LayoutManager Collapsed
Association 90%
assoc
«interface» ContainerPeer Collapsed
Delegation 80%
Composite 70%
FlowLayout Collapsed
17 MByte of 30 MByte allocated

Figure 7: Results of AWT analysis

We evaluated our approach by reverse engineering Java's *Abstract Window Toolkit (AWT)* library. Figure 7 shows a part of the annotated class diagram after the analysis. The detected pattern occurrences are marked by a rounded rectangle with the patterns name and the fuzzy belief of the annotation. Note, the class diagram is a rudimentary one, which does not contain associations, because they are not included in the source code; references are hidden. Associations that are detected by our rules are shown as Association patterns, e.g. between class Component and class Container.

The final analysis run with the modified patterns and adapted fuzzy values took about 2 minutes and 40 seconds for approximately 114,000 LOC on a 1.7GHz Pentium 4 machine with 1GB main memory. The first GoF-pattern was detected after 5 seconds. This shows first of all the scalablity of our process to real world systems and furthermore the success of the iterative approach. The re-engineer can interrupt the process and view intermediate results quickly even if the analyzed system is much more larger. One doesn't have to wait until the whole analysis is finished. The whole reverse engineering process of AWT including manual analysis of the source code and documentation lasted about 4 days, whereby the modifications and adaptations were made by a student involved in development of the prototype and therefore familiar with the rule's syntax and semantics. However, our experiences show that learning the syntax and using the tool can be mastered by novice users.

The screenshot shows at least four patterns detected by our algorithm, a Strategy between class Container and interface *L*ayoutManager with fuzzy belief 80%, a Composite between Container and Component with fuzzy belief 70% and a Bridge and a Strategy between Component and ComponentPeer with fuzzy belief 80%, each. All pattern, except of the last one, can actually be found in the source code and there exist no missed pattern in this part of the source code. The apparently false positive Strategy pattern in parallel to the Bridge pattern results from the fact that the two patterns are highly overlaid and we decided to define a Strategy as part of a Brigde.

7. CONCLUSIONS

Reverse engineering, in general, contains the problem that the systems to be analyzed consists of thousands and millions lines of code containing a large variety of different implementation styles. This paper presents an approach of fuzzy valued pattern matching applied to the reverse engineering of design patterns introduced by Gamma et al. We propose a semi-automated process to manage the large variety of implementation styles and to tune a pattern catalog to be able to analyze huge software sizes.

We are confident that our approach scales to even larger legacy systems. This is achieved by employing a rule set and an inference process that works quite "locally". In *bottom-up* mode, only the neighborhood of a certain trigger needs to be examined and in *top-down* mode, the provided context restricts the pattern matching task to a small fragment of the whole ASG. However, to achieve this, the pattern catalog needs to be carefully designed.

We employ somewhat imprecise pattern detection rules in order to cover a large number of implementation variants with a small number of simple rules. On the one hand, this impreciseness implies many false positives, but on the other hand the analysis is done more quickly. Alternatively, more precise rules produce less false positives but more rules may be required and the analysis takes longer, or may fail to detect unusual implementation variants. During the semi-automatic process, the re-engineer is able to tune the inference process by modifying, deleting or adding rules. More easily, the re-engineer may change the fuzzy values of certain rules and thereby lower or raise the influence of the corresponding sub patterns, e.g. due to their appropriateness for the current legacy system. Similarly, performance may be improved by raising the rule thresholds in order to restrict rule application to very reliable inputs. However, this may cause that certain design pattern occurrences are not found. Thus, tuning the fuzzy values and thresholds improves the inference process a lot. However, this is a tedious task requiring a lot of trial and error. A fuzzy learning component providing (semi-)automatic support for this tuning process is current work.

Note, the proposed re-engineering process is not restricted to design pattern detection. The approach is usable for various kinds of reverse engineering tasks and in many other application areas. For example, we currently investigate the usage of our fuzzy pattern definitions for the reverse engineering of story diagrams, i.e. programmed graph rewrite rules from legacy Java code.

8. REFERENCES

[1] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6th *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

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

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

[4] J. Jahnke and A. Zündorf. Specification and implementation of a distributed planning and information system for courses based on story driven modelling. In *In Proc. of Intl. Workshop on Software Specification and Design (IWSSD-9. Kyoto, Japan.*, pages 77–86. IEEE Computer Society Press, 1998.

[5] 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.

[6] 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.

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

[8] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance*, 5(4):181–204, December 1993.

[9] 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.

[10] A. Schürr, A. Winter, and A. Zündorf. The progres approach: Language and environment. In H. Ehrig, G. Engles, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2 - Application, Languages and tools.*, pages 487–546. World Scientific, Singapore, 1999.

[11] SUN Microsystems. *JavaCC, the SUN Java Compiler Compiler. Online at http://www.suntest.com/JavaCC.*

[12] 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.

[13] A. Zündorf. *Rigorous Object Oriented Software Development.* University of Paderborn, 2001.