Reverse Engineering with Fuzzy Layered Graph Grammars*

Jörg Niere[1], Albert Zündorf[2]

[1] Department of Mathematics and Computer Science, University of Paderborn,
Warburger Straße 100, D-33098 Paderborn, Germany
nierej@upb.de
[2] Institute of Software, Technical University of Braunschweig,
Postfach 3329, D-38023 Braunschweig, Germany
zuendorf@ips.cs.tu-bs.de

Abstract. Reverse engineering is a process highly influenced by assumptions and hypothesises 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 must support an interactive process, where the engineer is able to steer the analysis process by proving certain assumptions and hypothesises. 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 fuzzified graph transformation rules. We use layered graph grammars 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 lookup 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 library.

1 Introduction

Reverse engineering tries to recover design information from legacy source code. Simple reverse engineering approaches, using simple tools like `grep` and `perl`, have turned out as inappropriate for most of the interesting reverse engineering tasks, cf. [KP96]. 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. [MOTU93]. For the analysis of such abstract syntax graphs, graph grammars are well suited, cf. [Wil96, JZ98]. However, all these approaches face severe scalability and pattern complexity problems. First, the large size of some million lines of codes of typical legacy systems needs to be managed. Second, the flexibility provided by current programming languages give different programmers a lot of freedom to implement certain design elements in various ways. To deal with all these implementation variants, one needs a large number of variant design element detection rules. This large number of rules again creates a performance and complexity problem.

* This work is part of the Finite project funded by the German Research Foundation (DFG), projectno. SCHA 745/21.

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. [SWZ99, FNTZ98]. We allow superclasses 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 design element indicators. Instead of a large number of 100% precise descriptions of each possible implementation variant for a certain design element, we use a small number of somewhat imprecise detection rules, cf. [NSW+02].

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 detection 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 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 (version) used. To deal with this impreciseness more flexibly, this paper adds the usage of fuzzy techniques to the rules presented in [NSW+02] resulting in so-called *fuzzy layered graph grammars*. Each detection rule is equipped with a so-called *fuzzy belief* expressing the confidence of the reverse engineer in the rule within the current application context and a *threshold*, that restricts the rule application to cases with reasonable confidence. If a detection rule relies on intermediate results of other detection rules, the fuzzy belief of the generated design elements 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 detection rules. Section 4 discusses an example catalogue of detection rules for the “Gang of Four” design patterns [GHJV95]. How far our rule selection algorithm concentrates the efforts to certain local areas is discussed in Section 5. Section 6 shows how the reverse engineer may adapt the rules to a certain legacy system and Section 7 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 [HN90] 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 [Wil96]. 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

[Meh84]. Consequently, the approach is only useful to analyse not more than ten thousand lines of code.

Keller et al. [KSRP99] analyse behaviour 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 human engineer's domain and context knowledge. This contributes to scalability problems for the process enabled.

3 Fuzzy Reverse Engineering Rules

To model fuzzy reverse engineering rules, we adopt the Fujaba approach [FNTZ98, Zün01]. This approach has been formalized using a set-theory approach in [Zün01]. Main features of the Fujaba approach are a graph model with object-oriented features like inheritance, graph rewrite rules showing left- and right-hand side as a single graph in a UML collaboration diagram like notation, additional collaboration messages embedded within graph rewrite rules and UML activity diagrams as means for programmed graph rewriting. For the purpose of reverse engineering, we have adapted the Fujaba notation by introducing special shapes for design pattern artefacts and in this paper, by adding fuzzy beliefs to graph elements and reverse engineering rules, cf. Figure 2 and Figure 3.

For the purpose of this paper, the most important feature of the Fujaba graph model is the support for object-oriented inheritance. While this supports polymorphic method calls, reverse engineering rules mainly exploit, that superclasses may be used as wildcard node labels that match nodes of all direct and indirect subclasses. This mechanism has already been proposed in Progres, cf. [SWZ99]. For example, the Reference node r in Figure 2 may map on a host graph node with label Reference or with label MultiReference or with label ArrayReference, cf. Figure 1 and Figure 4. 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 superclass node labels.

Our reverse engineering rules search for certain clichés and pattern fragments and annotate them with design pattern markers (the oval shaped nodes). Thus, we usually employ a very complicated left-hand side while the right-hand side is just a copy of the left-hand side with one additional pattern marker node. In the Fujaba approach, a single graph is used to express both, the left- and the right-hand side of a rule. Nodes and edges belonging only to the left-hand side carry «destroy» markers (not employed for reverse engineering rules), nodes and edges belonging only to the right-hand side carry «create» markers (cf. Bridge node b and the attached edges in Figure 2) and nodes and edges without such markers belong to both sides of the rule. Note, our reverse engineering rules do not remove any graph elements and thus, the nodes and edges without

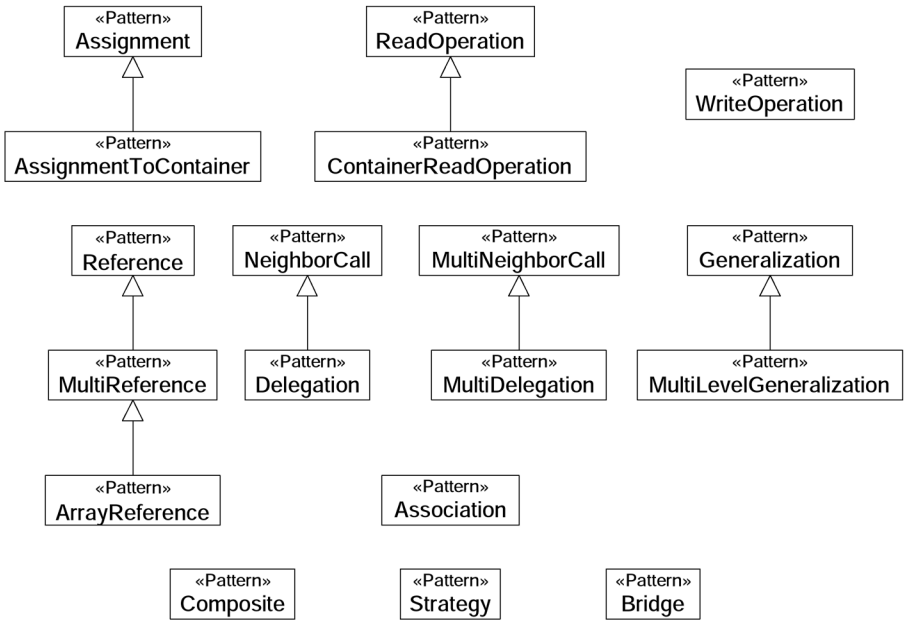

Figure 1 Pattern inheritance hierarchy

markers form the left-hand side of a rule and the right-hand side if formed by the union of the left-hand side and all «create» elements. In addition, our reverse engineering rules are restricted to the creation of a single new (oval shaped) design pattern annotation node and some links connecting this new annotation node to existing nodes via new links. For example, the rule of Figure 2 creates a single new Bridge annotation node b and an abtraction and an implementor edge to the annotated UMLClass nodes a and i, respectively.

Constraints on attribute values may be shown as boolean expressions within the “attribute compartment” of a node. General constraints may be shown within the rule as boolean expressions in curly braces.

Nodes or edges with dashed shapes / lines represent optional rule elements. This is a shorthand notation for one rule containing the dashed rule and another rule not containing the dashed element. Nodes with two stacked shapes match the set of all appropriate nodes in the host graph. Hollow lines with an arrow head are used to indicate OCL like navigational expressions that facilitate to express derived relationships between different rule nodes.

Crossed out nodes or edges represent negative application conditions. For example the crossed out Generalization node g2 in Figure 2 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. This will be discussed in chapter 4.

For the purpose of fuzzy reverse engineering rules, this paper extends the Fujaba rule notation with oval shapes for design pattern annotation nodes and with an implicit han-

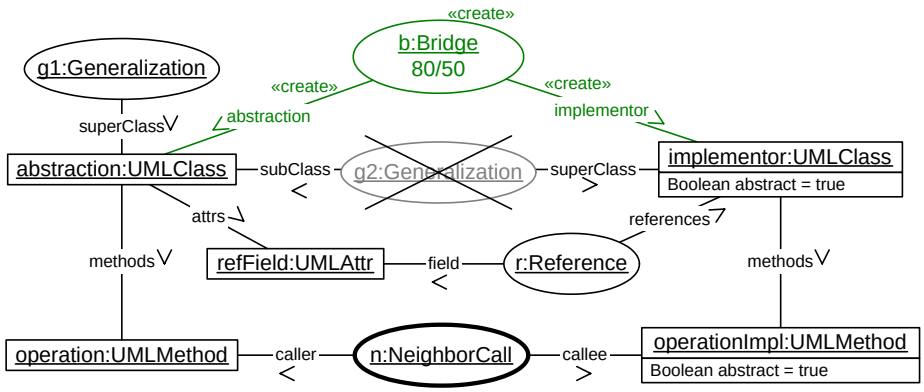

Figure 2 Bridge pattern rule

dling 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 2 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 successful applications. 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. The threshold is discussed below.

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. For example, if we apply the rule of Figure 2 and nodes g1, r and n are mapped to host graph nodes with fuzzy beliefs 60%, 70% and 65%, respectively, then 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 greater than 80%, the belief of the Bridge node would serve as an upper border for the whole rule: the overall belief cannot become higher than 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 assume that the fuzzy belief of one of the nodes that are matched by g1, r and n is very low, e.g. 42%. In that case the fuzzy threshold 50% of the example rule applies. As mentioned the second fuzzy value employed in our reverse engineering rules is a threshold that limits rule application to reasonable cases. If some of the intermediate results are very unreliable (have 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.

Note, possibly the creation of an annotation node could always trigger a rule which creates a new annotation node and this process could run, infinitely. In order to avoid such termination problems we adopt the approach of [RS95] proposing layered graph grammars for parsing of graphical diagrams. Basically, in a layered graph grammar the node kinds are separated in certain layers and each rule has either to reduce the number of elements or it has to add elements of a layer higher than the so far employed elements. This guarantees termination and allows to organize the parsing process more efficiently. In our approach, all design pattern annotation nodes are partially ordered according to dependencies derived from the rule set. For the rule set employed in this paper we have derived the dependency / support structure shown in Figure 4 as follows: An annotation node created by some rule gets a higher order than all (other) kinds of annotations that are employed in that rule. (The new annotation must not rely on an another annotation of the same kind). This dependency relation must not contain cycles. Thus, the application of a rule may only trigger rules on a higher level and the inference process will finally terminate.

4 A Rule Catalogue for the GoF Design Patterns

As stated in the introduction, reverse engineering is naturally an interactive process. Fully automated tasks 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` derivates, 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 *TogetherJ* try to detect, in addition to simple class diagrams with classes, attributes and methods, also relations between classes based on name conventions such as get-, set- and add-, remove- prefixes. The produced document is usually an 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 behaviour 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. [GHJV95] 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-pattern) for the patterns in-

troduced 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*"*[GHJV95 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, and collaborations between the classes.

Consequently, design patterns are highly suited to provide dependencies between certain parts of a software system and thus our reverse engineering approach focuses on the detection of GoF-patterns in Java source code. 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. Our approach is not limited either to GoF-patterns nor Java as implementation language because the analysis base is an abstract syntax graph representation of the source code and which patterns are analysed depends only on the rules.

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 re-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 system's design, etc. All parts are described in prose except the static structure, where Gamma et al. have used OMT class diagrams and sometimes the collaborations, where they have used collaboration diagrams. This informal definition is not sufficient for a tool supported reverse engineering process. In addition, the informal definition opens 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 layered graph grammars as described in the previous section. Each pattern is represented by one rule, whereas each rule creates exactly one new *pattern*-node with a certain fuzzy belief. Figure 2 shows the definition of the Bridge-GoF-pattern. We distinguish between two kinds of nodes. Nodes represented as ovals are pattern-nodes produced by other rules and nodes represented as rectangles are nodes of the abstract syntax graph created by an initial parser run. Abstract syntax graph nodes may contain attributes, such as the implementor node. The Bridge node b with a «create» marker represents the new pattern-node, which is created when the rule is applied. The left number inside the node is the fuzzy belief and the right number the threshold. This means that each pattern-node in the rule must have a belief higher than 50% when the rule is applied. The threshold can be refined by assigning a threshold to each pattern-node of the rule. For efficiency reasons, each rule, without the optional-, negative- and set-nodes, must form a connected graph. In theory, this is a severe restriction, but our experiences show that there always exists some kind of root node connecting different rule parts. In combination with so-called trigger nodes

(pattern-nodes with a thick border line), this condition allows us to provide a fast pattern matching algorithm introduced in the next section.

Our definition of the Bridge-GoF-pattern requires an abstraction class that is a superclass in an inheritance hierarchy represented by the Generalization pattern-node g1 in the upper left corner and that contains an attribute which is a Reference to another class implementor. The abstraction class must also contain a method, which has a NeighborCall to a method of the implementor class. In addition, the operationImpl method and the implementor class have to be abstract. The crossed out / negative Generalization node g2 in the middle means that the abstraction class must not be a (direct or indirect) subclass of the implementor class.

This definition omits details such as whether a generalization has to be a direct inheritance or an inheritance with some intermediate classes, and how a reference or a neighbour call looks like. Each of these sub-patterns has to be defined separately by its own rule. For example, Figure 3 shows the NeighborCall rule, which defines that a NeighborCall is a simple method call between two different classes related to each other via a Reference. The rule contains a path expression, which specifies that the method call must not be contained in a loop. (Otherwise we call it a MultiNeighborcall, cf. Figure 4).

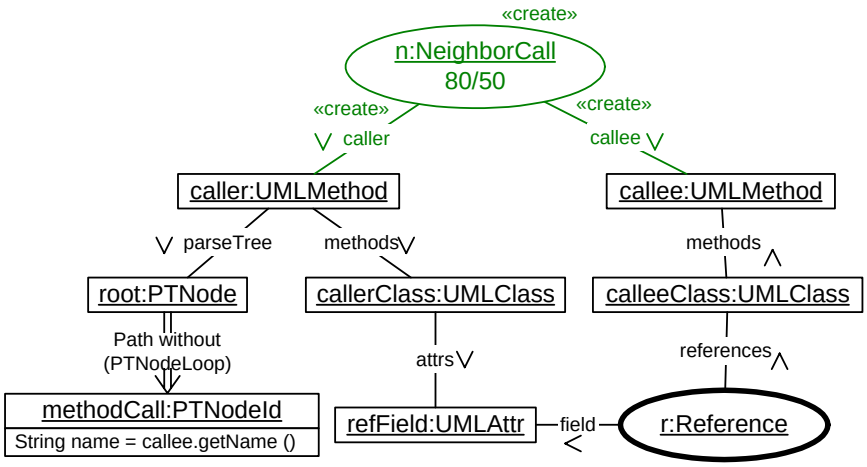

Figure 3 NeighborCall sub rule

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 and the arcs with an hollow arrow-head represent the inheritance relations between the pattern and abstract syntax graph nodes. For example, a MultiGeneralization pattern inherits from the Generalization pattern, where the Generalization means a direct inheritance and a Multi(Level)Generalization means an inheritance over more than one level. This is a recursive definition, because the MultiGeneralization depends on the Generalization and the pattern matching, described in the previous section, uses superclass labels as wildcards for subclass labels. In case of the Bridge pattern this means that the employed Generalization nodes might also be MultiGeneralizations, the NeighborCall might be a Delegation and the Reference might be a MultiReference or an ArrayReference.

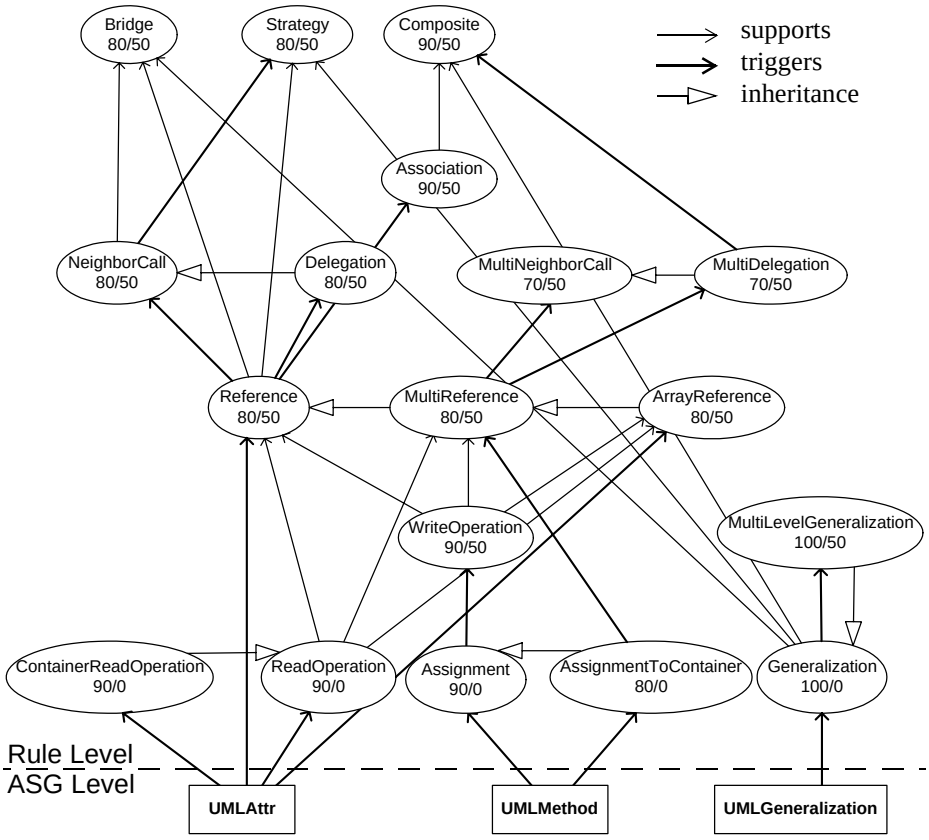

Figure 4 Pattern dependency graph including inheritance relations

Figure 4 shows a part of the pattern catalogue as an UML class diagram, where rectangle shaped classes represent abstract syntax graph elements. For pattern classes we use a certain stereotype and draw them as ovals containing the pattern's name, belief and threshold. The inheritance relations and dependencies are also shown, whereby the lines with the stick arrow heads denote the information flow.

The pattern catalogue of Figure 4 contains three GoF-pattern definitions, i.e. the Bridge-, the Strategy- and the Composite-GoF-pattern. All other sub-patterns represent different implementation variants. The whole catalogue including definitions for more than half of the GoF-patterns consists of about 75 patterns including the GoF-patterns itself. The catalogue is the result of three analysis processes and thus adapted to the specific analysed software systems.

5 A "Local" Rule Selection Algorithm

As mentioned above, the success of a semi-automated pattern-based analysis process highly depends on the production of reasonable intermediate results. Hence the subgraph isomorphism problem is NP-complete [Meh84], a smart pattern matching algo-

rithm is indispensable. The major requirement is that the algorithm is able to handle thousands and millions lines of code and either produces quickly a reasonable result or shows quickly that the used catalogue is not sufficient for the software system, i.e., the implementation variants are not covered by the rules.

We present only the ideas of the algorithm and how they are related to the theoretical approach of fuzzy layered graph grammars. A detailed description of the algorithm is described in [NSW+02].

As already mentioned, the sub-graph isomorphism problem is NP-complete, which lets brute-force algorithms fail, if the host graph, where a pattern should be found, is very large. Smarter algorithms, such as the algorithm contained in the Progres [SWZ99] and the Fujaba [FNTZ98] environment use the information in the pattern and try to traverse the host-graph using the pattern as a kind of map. For example, if two nodes are connected in a rule and one node is already bound to a node in the host graph, the algorithm traverses the edge to the second node and binds that node. If each rule contains an already bound "start" node, the complexity becomes polynomial or linear. In practice, the restriction that each rule contains at least one bound node causes no harm, because in our experience such a node is easily introduced.

Another optimization, especially concerning the production of reasonable intermediate results quickly, is to select a smart order in which rules are applied. Simple algorithms try to match an arbitrary rule of the rule base and in the failure case try another rule. Such an arbitrary choice can be optimized by a surrounding control-flow as in Progres or Fujaba. Our approach is to provide a smart automatic rule selection mechanism, i.e., a combined bottom-up, top-down approach, so that the re-engineer has not to specify an explicit control-flow.

To provide an optimized application sequence, we assume that the rules are partially ordered over their dependency relation, cf. Figure 4. This order also defines the layers of a layered graph grammar, cf. section 3 and [RS95].

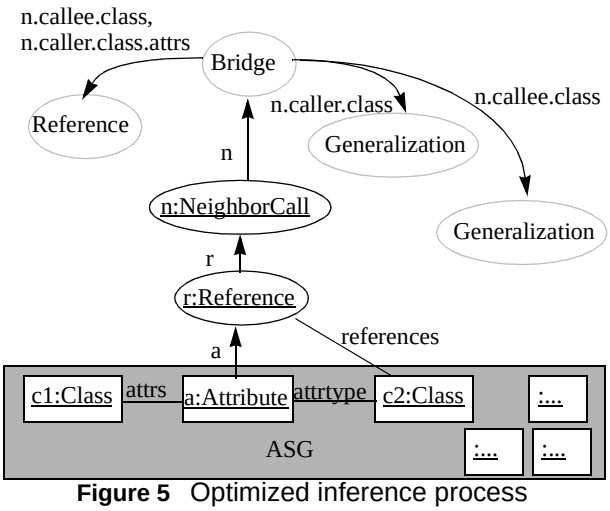

Figure 5 Optimized inference process

We call the application of rules an *inference process* following a fuzzy inference mechanisms. Figure 5 illustrates an example inference process. This inference process starts with parsing the source code in an abstract syntax graph (asg) representation. After this initial step, the algorithm starts in bottom-up mode, where it selects one of the asg-nodes, which is marked as a trigger in a rule, for example the Attribute node which serves as a trigger for Reference rules. In general the rule with the highest layer number is se-

lected to achieve a depth first search behaviour that quickly produces first results. In this case the Reference rule is applied next and a new node r:Reference is created. Next the NeighborCall rule creates a node n:NeighborCall, which triggers the Bridge rule. Since the Bridge rule depends on additional sub-pattern nodes, the algorithm switches into top-down mode and tries to establish or reject the hypothesis that a Bridge-GoF-pattern exists in the code. It is important that in top-down mode the algorithm provides a certain *context* for the detection of the sub-patterns, e.g. the Reference rule gets as context the callee class, that shall be the target of the Reference, and the set of attributes of the caller class, that may implement the Reference. Within the Bridge rule, these context nodes are accessible from the known NeighborCall trigger via n.callee.class and n.caller.class.attrs. If all required sub-pattern nodes can be found with sufficient fuzzy beliefs, the Bridge rule creates a new Bridge node. Note, each newly created node is added to the triggers used for the bottom-up mode. After a Bridge node is created or if one required sub-pattern cannot be found, the algorithm switches back to bottom-up mode and uses the triggers to continue the analysis.

Each time the algorithm is in bottom-up mode, the created pattern-nodes represent intermediate results. Choosing rules with high layer numbers, forces the algorithm to produce reasonable results, quickly, i.e., results about the existence of GoF-patterns are provided early instead of elicitating a large number of only minor important sub-patterns, first. Note, that the algorithm assigns initial fuzzy values to the pattern-nodes.

6 Adapting Rules

Our approach provides a semi-automated reverse engineering process, which is illustrated in Figure 6 depicted as a statechart. The first step in the process is to parse the source code and to create the abstract syntax graph including any kind of additional links, e.g. application to declaration links. Second step is, to load a pattern catalogue, which seams to be best adapted for the system. Afterwards the re-engineer can modify, add or remove rules, which is the start of the iterative process. A run of the analysis algorithm starts after all modifications are done and, as specified, the re-engineer can interrupt the algorithm each time it is in bottom-up mode, cf. previous section, or the algorithm halts when the analysis is complete.

The re-engineer is able to investigate the (intermediate) results and may adapt fuzzy values. After each modification, the fuzzy values are recalculated. This takes usually only a fraction of a second, because the recalculation stops if no fuzzy value changes. In case of a modification of a rule, the algorithm has to start at the beginning, again. This point provides many possibilities for improvements we are currently investigating. The reverse engineering process ends when the re-engineer decides that the produced results are sufficient.

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

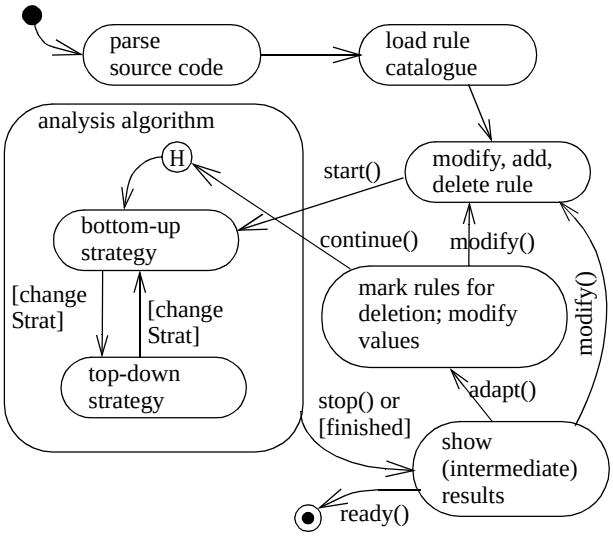

Figure 6 Reverse engineering process statechart

Figure 7 shows a part of the annotated class diagram after analysing Java's Abstract window toolkit (AWT) library. The detected patterns and subpatterns are marked by an icon at the top of the pattern name and they show the fuzzy beliefs for the detected pattern annotations. 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 rule base 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 for approximately 120 KLocs on a 1GHz Pentium 4 machine with 1GB main memory. The whole reverse engineering process 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 LayoutManager 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 analysed consists of thousands and millions lines of code containing a large variety of different

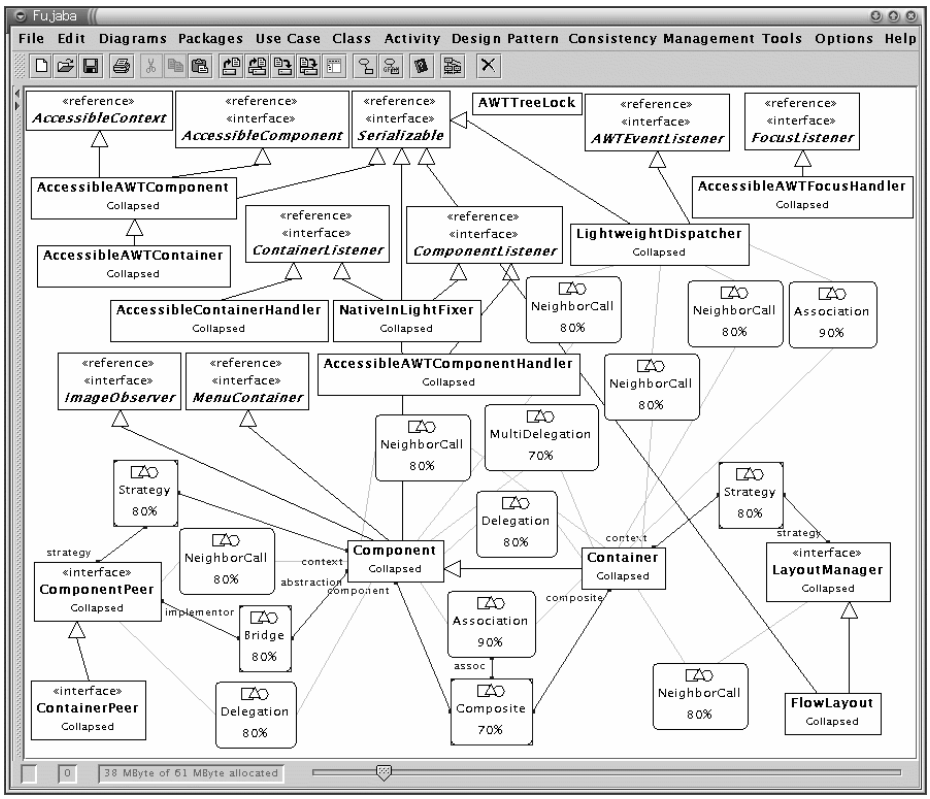

Figure 7 AWT analysis result as a class diagram

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-catalogue to be able to analyse a huge software size.

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 neighbourhood 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 abstract syntax graph. However, to achieve this, the pattern catalogue 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 and more complicated 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 modifying, deleting or adding rules in the rule catalogue. 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 fuzzy layered graph grammar approach is not restricted to design pattern detection. The approach is useable for various kinds of reverse engineering tasks and in many other application areas. For example, we currently investigate the usage of our fuzzy layered graph grammars for the reverse engineering of story diagrams, i.e. programmed graph rewrite rules from legacy Java code.

8 References

[FNTZ98] 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.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HN90] M. T. Hanrandi and J. Q. Ning. *Knowledge Based Program Analysis*. IEEE Transactions on Software Engineering, 7(1):74–81, IEEE Computer Society Press, 1990.

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

[JZ98] J.H. Jahnke and A. Zündorf. *Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment*. Technical Report tr-ri-98-201, University of Paderborn, Paderborn, Germany, 1998.

[KP96] 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.

[KSRP99] R.K. 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.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1st edition, 1984.

[MOTU93] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. *A Reverse Engineering Approach To Subsystem Structure Identification*. Journal of Software Maintenance, 5(4):181–204, John Wiley and Sons, Inc., December 1993.

[NSW+02] J. Niere, W. Schäfer, J.P. 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, May 2002.

[RS95] J. Rekers and A. Schürr. A Graph Grammer Approach to Graphical Parsing. IEEE Symposium on Visual Languages (VL'95), Darmstadt, Germany, 1995.

[SWZ99] A. Schürr, A.J. 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.

[Wil96] L.M. 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.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, Germany, 2001.