

# User-Driven Adaption in Rule-Based Pattern Recognition

Jörg Niere  
Software Engineering Group  
Department of Computer Science  
University of Siegen  
Hölderlinstr. 3  
57068 Siegen, Germany  
joerg.niere@uni-siegen.de

Matthias Meyer, Lothar Wendehals\*  
Software Engineering Group  
Department of Computer Science  
University of Paderborn  
Warburger Straße 100  
33098 Paderborn, Germany  
[mm/lowende]@upb.de

## Abstract

Today, in software intensive projects a huge amount of the budget flows into the analysis of the already existing system. The reason for the high costs results mainly from the fact that analyses are often made manually or with automatic tool support, which is inappropriate for analyzing large systems. Semi-automatic analysis approaches usually use a notion of fuzziness to overcome this limitation, but inherit the problem of selecting appropriate initial values.

In this paper we present an approach to adapt the initial values of our semi-automatic reverse engineering process. We provide the reverse engineer with accuracy information for results produced by a rule-based inference algorithm. Based on the changes of the results done by the reverse engineer we automatically adapt a credibility value of each rule, which previously has been used to compute the accuracy of the result. The adaption fits seamlessly into our overall analysis process. First tests show that it is suitable for the calibration of our fuzzyfied rule-based pattern recognition approach.

## 1. Introduction

The work presented in this paper is part of our reverse engineering project to analyze large software systems. Within the project we recover design information, i.e. design patterns [8], through a tool supported semi-automatic pattern-based recognition process. Our process is based on an analysis of a system's source code, because documentation may not be available or may have become obsolete. Recognizing design patterns instances in existing software systems helps the reverse engineer to understand the system. However, the approach is not limited to design patterns, but it is also suitable for the detection of other kinds of software patterns, e.g. implementation patterns [16], architectural patterns [3] or patterns from pattern-languages [5].

In addition, our approach is applicable for coarse-grained and fine-grained analysis activities. We also challenge the ability to analyze large systems with more than 100 kLOC, cf. [21], where automatic approaches fail. In order to handle the large search space we introduce a notion of fuzziness to our approach and we assess the accuracy of its results.

Furthermore, in our approach of a tool supported semi-automatic recognition process we highly involve the reverse engineer. Our process consists of basically three steps or-

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

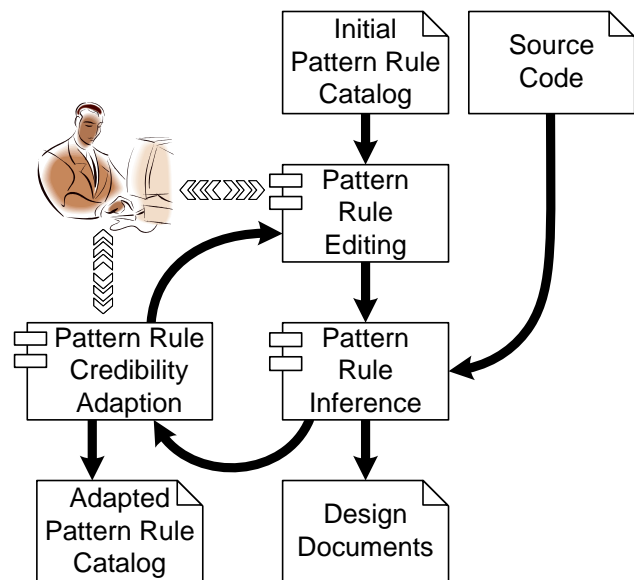


Figure 1: The pattern recognition process

ganized in a cycle, cf. Figure 1. An analysis of a system is performed iteratively, i.e., the cycle is passed through several times.

The process starts with an initial catalog of pattern specification which may be edited by the reverse engineer in the *Editing* step. We developed a formal pattern specification language based on graph grammar theory [20], where each pattern is defined as a single graph-rewrite-rule, a so-called pattern rule. Within different iterations of our process the *Editing* step is used to adapt the pattern rules to the system under analysis with respect to, e.g. specific programming styles used in the implementation.

In the *Inference* step, the rules are automatically applied by an inference algorithm that produces meaningful results early and not only after a complete analysis of the system. The reverse engineer may interrupt the inference to inspect the results produced so far and may then decide to continue or abort. In order to support the reverse engineer in this decision we assess the accuracy of the results and assign an accuracy value to each recognized pattern instance [21]. The accuracy value is computed based on credibility values which are attached to the pattern rules. Unfortunately, the

initial credibility value of a rule can only be estimated by the reverse engineer specifying the rule. Thus, so far the precision of the credibility values highly depends on the personal experience of the reverse engineer or previous analyses.

To overcome this limitation we developed a tool supported semi-automatic approach to help the reverse engineer to improve the precision of the credibility values, which is presented in this paper. We extend our process with the *Credibility Adaption* step, in which the reverse engineer is allowed to manually override the accuracy values computed for the results of the previous *Inference* steps. Based on the accuracy values changed by the reverse engineer we automatically adapt the credibility values of the corresponding pattern rules. The adaption underlies simple statistics, which results in a very fast algorithm and does not produce much overhead in the highly interactive analysis process. The effects of the adaption are immediately transferred to the results produced during previous *Inference* steps<sup>1</sup>.

In the following Section we present related work. In Section 3 we describe our rule-based pattern recognition reverse engineering approach in more detail before Section 4 presents our newly developed approach to the automatic adaption of the credibility values. Section 5 gives a summary and points out future work.

## 2. Related Work

Design recovery approaches mainly focus on automatic analysis and information extraction using pattern-based analysis methods. One of the first general approaches of analyzing programs has been the event-based system presented by Harandi and Ning [9], detecting program plans. Wills [28] uses attributed flow-graph-grammars to detect program plans with focus on the combination of control- and data-flow analysis. Radermacher [23] uses general graph-grammars and the Progres system [31] to detect design patterns in source code. All these and most other pattern-based reverse engineering approaches focus on automatic analysis, e.g. [10]. Therefore the approaches are not suitable for large software sizes with respect to a detailed degree of analysis or produce many false-positives. Especially the approaches in [23] and [28] inherit the NP-complete subgraph isomorphism problem [17].

Interactive approaches such as Rigi [18] and Gupro [6] provide generic frameworks, e.g. an interactive graph browser that have to be adapted to certain reverse-engineering exercises. Traversing information within the framework's representation back to the original code is not supported or only hard to achieve.

An interactive approach for reverse engineering database schemas and migrating its data presents Jahnke in [11]. The approach allows for handling uncertain and incomplete knowledge and uses possibilistic reasoning with so-called Generic Fuzzy Reasoning Nets. In addition to the original schema and the actual data, the approach uses SQL code as basic facts. To extract such facts Jahnke uses graph-grammars and the Progres system. In the database reverse engineering domain a small number of rules is sufficient to extract base facts automatically. In case of design recovery, we would

<sup>1</sup>In later iterations of our process when an adaption of the pattern rules is no longer necessary the *Editing* step is often skipped. After an inference the *Credibility Adaption* step may be skipped as well to directly start a new iteration with (further) adaption of the rules.

need a very high number of rules and therefore the approach is not suitable for the analysis of large systems.

A general problem in fuzzy reasoning based systems is the choice of the initial reasoning values. Unfortunately, those values could only be estimated if the source code that should be analyzed is known. A solution to overcome this problem is to learn or adapt the values [4] based on learning exercises, which is called machine learning. Popular approaches are gradient descent approaches [27] in combination with back-propagation [7]. Jahnke and Strebin [12] use fuzzy neural nets that are a specific gradient descent approach with back-propagation to learn the fuzzy values during the interactive reverse engineering process. It turned out that constructing the learning exercise takes much time even using small generic fuzzy reasoning nets.

Machine learning and especially neural nets are often used in general pattern recognition approaches, e.g. in detecting images, faces or orientations of artifacts [22], [15], [2], [1], [24]. Unfortunately, a premise to use neural nets is that the structure of the net for which the values should be learned does not change. The idea of our approach to design recovery is that a reverse engineer iteratively determines the specific rules that analyze the specific system appropriately. Our approach is based on the idea of rapidly changing pattern rules, especially in the beginning of the analysis process. Therefore machine learning approaches are not suitable for our approach.

## 3. Rule-Based Pattern Recognition

Patterns and especially design patterns [8] are usually informally described, allowing the programmer a wide variety of implementations. In order to support pattern recognition by tools patterns have to be formally defined. We developed a formal pattern specification language based on graph grammars. The rule definition is graph based, which means that 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 programming language or paradigm. We define patterns as graph-rewrite-rules with respect to the abstract syntax graph (ASG) representation of a program. We call such graph-rewrite-rules pattern rules.

Analyzing a system starts with partially parsing the system's source code into an ASG. Then the pattern rules can be applied to the ASG. Parts of the ASG that are currently not available are parsed incrementally during the rule application. The analyzed system is visualized as a UML class diagram with annotations indicating found pattern instances. As an example we analyzed the Java Abstract Window Toolkit (AWT). Parts of the library are shown in Figure 2. We found three design pattern implementations within this part, namely a *Composite*, a *Strategy* and a *Bridge* design pattern. We found further auxiliary patterns that are not displayed for readability reasons except for the *ItoN\_Association* pattern. See [21] for more details detecting associations.

Each annotation for the found design patterns - shown as a dashed oval - has a name and a value between 0% and 100% representing the accuracy of the result. An annotation is linked to class diagram elements, indicating relevant parts of the pattern such as the *Component* and *Composite* classes of a *Composite* design pattern, cf [8]. On the basis of the

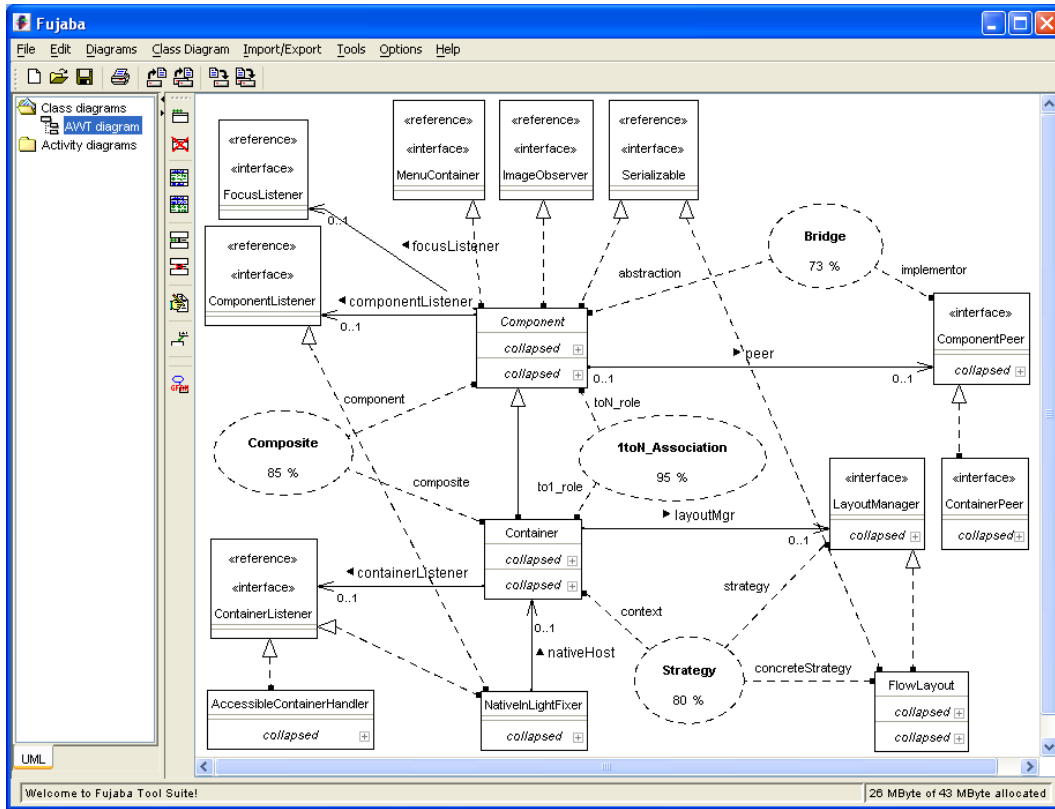


Figure 2: Annotated UML Class Diagram

annotation’s accuracy value the reverse engineer can review the result. This is done by changing the accuracy value.

### 3.1 Pattern Rule Definition

A graph-rewrite-rule is composed of a left hand side and a right hand side. The left hand side of a graph-rewrite-rule consists of a subgraph that has to be isomorphically bound to a host graph. The right hand side of a graph-rewrite-rule consists of the left hand side and modifications to the subgraph, i.e., nodes and links to be created and/or deleted. The rule is applied by first finding an isomorphic binding and then modifying the subgraph.

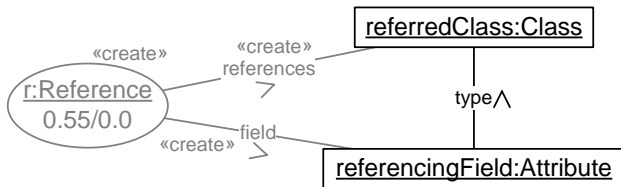


Figure 3: Reference Rule, Variant 1

Pattern rules are a special kind of graph-rewrite-rules. In each pattern rule only one node, i.e. the annotation node, and some links that connect the annotation node to the bound subgraph are created. Figure 3 shows an example for a simple pattern rule. In the notation used, the left hand side of the rule is defined by the black nodes and edges. The right hand side consists of the left hand side and the mo-

difications defined by the gray node and the links with the stereotype «create». Instances of the ASG-nodes are represented by rectangular shapes, annotations are represented by oval shapes.

This pattern rule defines a reference between two classes. The left hand side consists of two nodes, an attribute named `referencingField` and a class named `referredClass`. They are connected by a link named `type`, which means that the attribute `referencingField` is of type `referredClass`. The oval shaped node `r:Reference` is the annotation which identifies the subgraph of the ASG as match of a reference. When the rule is applied this node including the edges are created and thus, enrich the ASG semantically. Thus, we can conclude that the result of a pattern rules’ application is an annotation within the ASG.

The annotation node has two values, the credibility on the left value and the threshold on the right. The credibility value represents the ratio of accurately recognized pattern instances to all pattern instances recognized by the rule. Here in this example the credibility value of the pattern rule is 55%. That means that the reverse engineer who has defined the rule expected 45% of all recognized instances to be false-positives. The pattern rule looks for an attribute referencing a class. It would match for example an attribute referencing a container class, i.e., a to-one reference to a container such as a list. In most cases this is only one way to implement a to-many reference to another class. Objects of the other class are the elements within the list. In a technical view of the system, a to-one reference to the container would

be correct. In a design view a to-many reference to the class of the container elements would be correct and the to-one reference would be a false-positive.

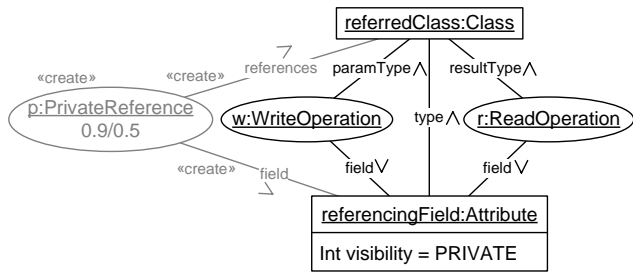


Figure 4: Reference Rule, Variant 2

Another implementation example for a reference caught by a pattern rule is depicted in Figure 4. Here a private attribute with read and write access methods should be found. Note, that the annotation nodes created by one rule can be used by another rule. In this example the access methods are recognized by other rules so annotation nodes are used within this rule. Thus, the application of the `p:PrivateReference` rule depends on the successful application of two other rules, namely `w:WriteOperation` and `r:ReadOperation`.

This pattern rule has a higher credibility value than the first one. The reverse engineer expects an instance found by this rule to be more likely a reference, since there are access methods using the referred class as parameter or result type. Container access methods usually do not have the container class as parameter type but the class of the container elements, cf. the implementation of Java Beans [25].

The threshold of a pattern rule influences the visualization of the results. If a certain annotation has an accuracy value lower than the threshold of its corresponding pattern rule, the annotation will not be displayed. However, the reverse engineer may view them for investigation, explicitly.

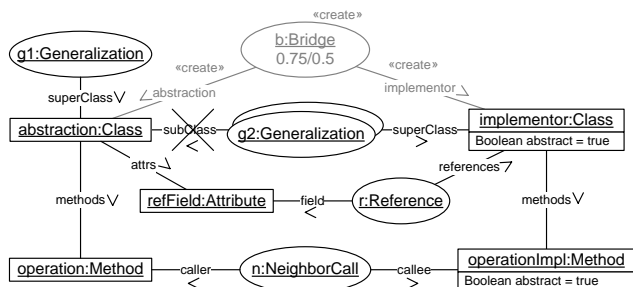


Figure 5: Bridge Rule

Figure 5 shows a pattern rule for the *Bridge* design pattern [8]. A *Bridge* decouples an abstraction from its implementation so that both can vary independently. This is done by putting the abstraction and the implementation into two different class hierarchies. This fact is expressed within the pattern rule by the `g2:Generalization` annotation with the double oval. The double oval means that during subgraph matching a set of `Generalization` annotations can be bound to the pattern rule's element. The crossed link between `g2:Generalization` and the class `abstraction` prohibits that any of the `Generalizations` has annotated the abstracti-

on class as a subclass. So the implementation hierarchy is independent from the abstraction hierarchy.

The gap between the hierarchies is bridged by a reference between the two super classes. Methods from the abstraction hierarchy are implemented within the implementation hierarchy, thus method calls are directed from the abstraction to the implementation. The `g1:Generalization` annotating the abstraction class as super class indicates that there are sub classes of the abstraction class.

Therefore, higher level pattern rules are formed by combining lower level pattern rules. This forms a net of dependencies between the pattern rules which is described in the following together with the application of the rules.

### 3.2 Pattern Rule Application

The pattern rule inference algorithm needs information about dependencies between the rules to minimize rule application failures. The dependencies are computed from the pattern rule definitions and stored in a so-called Pattern Dependencies Net (PDN) as shown in Figure 6.

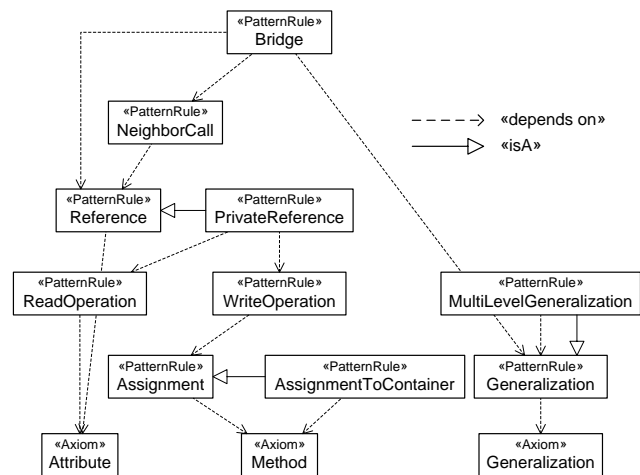


Figure 6: Pattern Dependencies Net

A pattern rule A depends on another pattern rule B, if an annotation node created by rule B is used within the definition of pattern rule A. As you can see in Figure 4 the pattern rule `PrivateReference` depends on the pattern rules `WriteOperation` and `ReadOperation` which is represented in the PDN in Figure 6 by `«dependsOn»`-links.

In contrast to the `«dependsOn»` relationship between pattern rules, the `«isA»` relationship can not be computed. It is defined by the reverse engineer to express that one pattern rule has the same semantics as another pattern rule. In our example the `PrivateReference` pattern rule from Figure 4 `«isA»` special kind of the `Reference` pattern rule depicted in Figure 3. For the application of the `Bridge` pattern rule for example this means, that instead of a `Reference`-annotation a `PrivateReference`-annotation can also be used.

In addition to pattern rules the PDN contains so-called axioms as nodes. Axioms are elements from the ASG. In classical rule-based systems the rules are leveled according to their topological order within the PDN and applied level by level. The inference begins with pattern rules directly depending on axioms and ends with rules on the highest level such as the `Bridge` rule, cf. Figure 5. These classical

algorithms have the problem, that meaningful results from high level rules are produced late in the inference process. We implemented a smarter algorithm that produces relevant results early to aid an interactive process.

Another problem in subgraph matching algorithms is the NP-completeness of subgraph isomorphism [17]. We deal with this problem by computing a context, i.e., a set of objects. A context could include axioms or annotation objects created by other rule applications and can be computed based on the structure of the rules, only. The pattern rule is then applied in the context which reduces the time complexity to polynomial and in average cases to linear time. The algorithm is explained in detail in [20].

Figure 2 shows the result of the inference algorithm. The reverse engineered system is represented as a UML class diagram with annotations indicating pattern instances. Each annotation has a name and an accuracy value. Annotations with an accuracy value lower than the threshold of the corresponding pattern rule are not displayed.

## 4. User-Driven Adaption

The inference process as shown above can be done automatically except for the pattern rule catalog definition. Since the process is designed as an iterative one, the user is encouraged to interact with the process. We achieve this in two different ways. The reverse engineer can start the inference with an initial pattern rule catalog and modify the rules during the iterations, cf. Figure 1. If the results do not have an accuracy the reverse engineer has to investigate all results, here annotations, in order to determine their correctness.

Each of our pattern rules has a credibility value and a threshold, initially estimated by the reverse engineer. We use the credibility values to calculate an accuracy value for each annotation created by the pattern rule. More precisely, we have switched from binary decisions, where the rule matches or not, to multi-valued decisions such as provided by fuzzy-sets [29, 30].

The accuracy value is the basis for the investigation of the annotations by the reverse engineer. If the credibility values are chosen appropriately, a low accuracy value indicates an unreliable result, whereas a high accuracy value indicates a reliable result. Therefore the reverse engineer does not have to investigate all results, but only problematic or interesting ones according to the values. However, the problem of estimating initial credibility values still remains.

During the investigation the reverse engineer will probably disagree with one or more annotation's accuracies as the annotations are false-positives or correct for sure. Then the reverse engineer may change the accuracy values. Since we use the thresholds of the pattern rules to filter the annotation results displayed to the reverse engineer, it might be the case that he/she wants to accept a filtered annotation or to reject a displayed annotation.

In our approach all the changes of the reverse engineer made during the investigation are used to automatically adapt the credibility values and the thresholds of the pattern rules. The adaption leads to better adjusted credibility values and thresholds and therefore will produce better results in further analyses. In the next sections we explain how the accuracy values are calculated and how the credibility values of the rules are adapted on the basis of the reverse engineer's evaluations.

## 4.1 Accuracy-Value Calculation

The calculation of an annotation's accuracy value is a downstream operation after the inference algorithm has terminated or the reverse engineer has interrupted it. Separating the calculation of the accuracy values from the inference process has the benefit, that the calculation could be used to recalculate the accuracy values. A recalculation is necessary when the reverse engineer has modified some accuracy values.

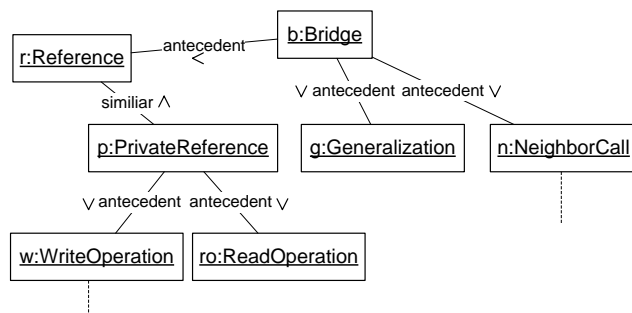


Figure 7: Net of annotations created by inference

The result of the inference algorithm is an annotation-net represented as an object-oriented graph such as shown in figure 7, where the graph nodes represent the annotations and the edges in the graph represent antecedent and similar relations. In object-oriented terms, the antecedent relations are instances of the pattern rule dependencies and the similar relations are instances of the generalizations between pattern rules in the pattern-dependency-net depicted in Figure 6. Note that the directions of similar like correspond to the direction of the generalization relation in the PDN.

For the calculation of the accuracy values the object-oriented graph shown in Figure 7 is interpreted as a Fuzzy-Petrinet (FPN) [14], cf. Definition 1.

DEFINITION 1. A Fuzzy-Petrinet is a tuple  $FPN = (P, T, F, cr, th, uac, cac, ac)$  with

- $P$  is a finite set of places
- $T$  is a finite set of transitions
- $F \subset (P \times T) \cup (T \times P)$  is the set of edges
- $cr : P \rightarrow [0..1]$  assigns the credibility value
- $th : P \rightarrow [0..1]$  assigns the threshold
- $uac : P \rightarrow [0..1]$  is the user defined accuracy value, the function is partial
- $cac : P \rightarrow [0..1]$  is the calculated accuracy value
- $ac : P \rightarrow [0..1]$  with  $\forall p \in P$  is

$$ac(p) = \begin{cases} uac(p) & : \text{if defined} \\ cac(p) & : \text{else} \end{cases}$$

The FPN in Definition 1 is used to calculate the accuracy values. It is a place-transition net, where the function  $cr$  and  $th$  assign the pattern rule's credibility and threshold values to the corresponding places. The function  $ac$  assigns the actual accuracy value to a place. The actual accuracy value is calculated by the function  $ac$ , which is defined as  $uac$  if the accuracy value has been modified by the reverse

engineer, or as the function  $cac$  for the calculated accuracy value.

The transformation of the object-structure produced by the inference algorithm into an FPN according to Definition 1 is nearly a one to one transformation. Firstly, for each annotation in the object-structure exists a corresponding place  $p \in P$  and functions  $cr$  and  $th$  assign the credibility and threshold values to the places. Secondly, for each antecedent link exists a corresponding transition  $t \in T$ . In addition, edges connecting the transitions with places form a subset of  $F$  with respect to the antecedent links. Thirdly, similar links produce additional edges in  $F$  connecting places to transitions that correspond to the appropriate antecedent link. Figure 7 shows a cut-out of an object-structure and the appropriate Fuzzy-Petrinet is shown in figure 8, whereas the figure already includes the calculated accuracy values. Note that the similar link would have had no influence if the direction was vis-versa.

DEFINITION 2. A Fuzzy-Truth-Token is defined as the function  $ftt : T \rightarrow [0..1]$  with  $\forall t \in T$  is

$$ftt(t) = \max(\{ac(p) | (p, t) \in F\})$$

The function  $cac$  that assigns the calculated accuracy value to each place in the Fuzzy-Petrinet is determined in two phases, cf. [14]. In the first phase each transition calculates its so-called fuzzy-truth-token, cf. Definition 2. The Fuzzy-Truth-Token is the maximum of all accuracy values at places reachable over incoming edges of the transition; so-called preceding places.

In cases where no similar link exists in the object-structure, the corresponding transition has only one incoming edge. If a similar links exists, the maximum accuracy value is taken as Fuzzy-Truth-Token. A similar link means that at least two pattern instances have been found annotating the same objects and the corresponding pattern rules participate in an inheritance relation. An example is given by the pattern rule in Figure 3 and the pattern rule in Figure 4, which represent different types of references between classes.

DEFINITION 3. The calculated accuracy value for a place is given as  $\forall p \in P$  let  $m_t(p) = \min(\{ftt(t) | (t,p) \in F\})$  and

$$cac(p) = \begin{cases} cr(p): & \neg \exists t \in T \text{ with } (t, p) \in F \\ \min(\{cr(p), m_t(p)\}): & \text{else} \end{cases}$$

In the second phase, the calculated accuracy value can be determined which is the minimum of all Fuzzy-Truth-Tokens at the transitions reachable over incoming edges and the credibility value of the place itself, cf. Definition 3. In cases where the place has no incoming edge, i.e. axioms, the  $cac$  is the credibility value of the place itself. In this definitions the threshold of a place is not taken into account. This ensures that unreliable results can be investigated by the reverse engineer as well. It might be the case that the accuracy value is often only a little bit lower than the threshold. This fact is swept away if the accuracy value is, for example, set to zero if it is lower than the threshold. On the other hand setting the accuracy value to zero enabled us to provide a faster calculation algorithm, because subsequent Fuzzy-Truth-Token have not to be calculated. In our approach we use the thresholds only to filter the results displayed to the reverse engineer.

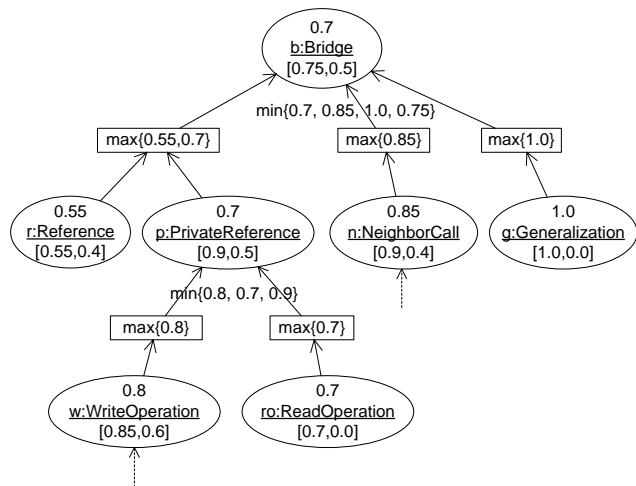


Figure 8: Example Fuzzy Petrinet created from object-structure in Figure 7

Figure 8 shows an example for the calculation. The name of the place is shown in the middle of the oval, e.g. **b:Bridge**. The calculated accuracy value after evaluating the whole FPN, i.e. after the FPN is stable, is shown above the name. Thus, the number represents the result of function  $ac$ . Since the reverse engineer has not changed the accuracy values the values are identical to the calculated accuracy value, cf. function  $cac$ . Below the name we show the credibility value and the threshold of the corresponding rule. The calculation of the Fuzzy-Truth-Token, i.e., the maximum function is shown in the rectangle of a transition. The calculation of the  $cac$  function, i.e., the minimum is shown at the incoming edges of a place.

In our approach the stability property of the Fuzzy-Petrinet, in means of iterating phase one and two and stopping when the accuracy values do not change any longer cf. [14], is easily provable. Our approach only allows the reverse engineer to define a special kind of cycle in the pattern-dependencies-net. Only cycles that do not overlap and cycles that contain one generalization are valid. In addition, all pattern rules participating in a cycle have to fulfill a reduction property that ensures a finite number of rule applications, cf. [19]. These special requirements ensure that the FPN created out of the object-structure is cycle-free and consequently, the iteration of phase one and two must stop. More precisely, the maximum number of iterations is equal to the longest path in the FPN starting from an axiom following outgoing edges to a place without outgoing edges.

## 4.2 Adaption

After calculating the accuracy values for the annotations, the reverse engineer can investigate the annotation results. First of all, the reverse engineer will investigate annotations that correspond to design pattern instances such as a *Composite* or *Bridge* pattern. Later he/she will also investigate the inference for a certain annotation or will have a closer look on situations where no inference has taken place.

The reverse engineer takes an annotation's accuracy value as indication for further investigation and will change some values that represent the accuracy of the actual result more appropriately. On the assumption that the modified accura-

cy values are more appropriate, the modifications are a basis for the adaption of the credibility values.

The reverse engineer is also able to accept a filtered result or reject a result produced by the inference algorithm. This situation raises when the threshold of the pattern rule is too high or low, respectively. Therefore we take such interaction by the reverse engineer to adapt the threshold of a pattern rule.

As formal basis for the adaption of the credibility values as well as the thresholds, we use a simple statistic model in means of an average computation. For the average calculation we use the calculated accuracy value as actual value. In case of the adaption of the credibility value, we take as nominal value the value corrected by the reverse engineer. In cases where the reverse engineer only accepts or rejects a result, the nominal value is the threshold of the corresponding pattern rule. In both cases we store the new value as a virtual value in sets  $ac_v(p)$  or  $th_v(p)$  for each place  $p$  in the Fuzzy-Petritnet.

#### 4.2.1 Accuracy Value Changes

When the reverse engineer corrects the accuracy value of an annotation, the corrected value is the nominal value, whereas the calculated accuracy value is the actual value. For example in Figure 9 the reverse engineer corrects the accuracy value of the **b:Bridge** annotation from 70% to 90%. Since each annotation maps to exactly one place in the FPN, the corrected accuracy value of 90% is stored as a virtual value in the set  $ac_v(b:Bridge)$ .

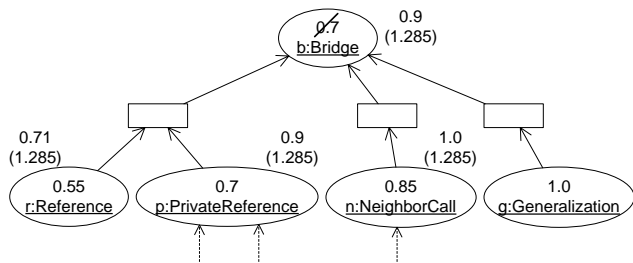


Figure 9: Increase of an accuracy value

In addition to the virtual value added to the **b:Bridge** place we also add virtual values to preceding places. As basis we take the calculation of function  $cac$ , cf. Definition 3, which means that we raise necessary accuracy values to force a correct local calculation of the places'  $cac$  value. In the situation in Figure 9, we have to raise the accuracy value of the **r:Reference**, the **p:PrivateReference** and the **n:NeighborCall** annotation, because their calculated accuracy value is lower than 90%. The accuracy value of the **g:Generalization** will not be corrected, because the calculated accuracy value is 100% and therefore the accuracy value of the **b:Bridge** can not be affected by the minimum function. We multiply all preceding accuracy values by 128,5%, because this is the relative increase given by the reverse engineer. The relative values are shown in parentheses below the virtual accuracy values. The virtual accuracy values are stored in the  $ac(p)$  sets corresponding to the annotations. In cases where the accuracy value is higher than 100%, e.g. the **n:NeighborCall** with originally 102%, we cut the virtual value to 100%. In this situation, the correction of the reverse engineer results in four virtual values at different places.

We do not propagate the virtual values through the whole annotation structure, because this results in the effect, that facts lower in the dependency hierarchy, which many annotations depend on, get too many virtual values. A better solution is that the reverse engineer traces the certain inferences and corrects an accuracy value deeper in the structure.

In addition, changes of an accuracy value - not the virtual values - result in a recalculation of the FPN. For example, if the reverse engineer corrects the accuracy value of the **p:PrivateReference** annotation in Figure 9 from 70% up to 75%, this has the effect, that the accuracy value of the **b:Bridge** also becomes 75%.

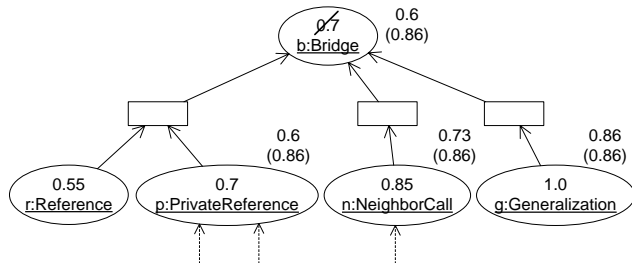


Figure 10: Decrease of an accuracy value

Figure 10 depicts the situation that the reverse engineer decreases the accuracy value of the **b:Bridge** annotation down to 60%, which results in a corresponding virtual value. As in the previous case, the correction results also in virtual values for preceding places. In the situation shown in Figure 10 all places except the **r:Reference** place get a virtual value, because the **r:Reference** place has a calculated accuracy value of 55%, which is lower than 60%.

Note, such a choice of virtual values, especially the calculated ones, has the characteristics that they converge to 0 slower than to 1, because of the cut of values higher than 1. In our approach we take this as a benefit, because therewith decreasing a value has not as much influence as increasing. Even in combination with thresholds, a high influence of decreasing values often results in values below the threshold that will consequently be filtered.

#### 4.2.2 Accepting or Rejecting Results

The threshold of a pattern rule filters annotation results with low accuracy values. Those results are usually not presented to the reverse engineer except he/she requests them, explicitly. In contrast to the correction of the accuracy value the reverse engineer is only allowed to reject a result or to accept a filtered result. Accepting a non-filtered result or rejecting a filtered result is not supported. When the reverse engineer rejects or accepts a result, he/she influences the adaption of the threshold of the certain pattern rule.

Figure 11 shows two examples where in the top of the Figure the reverse engineer accepts a filtered **b1:Bridge** annotation. The bottom of the Figure shows an example where the reverse engineer rejects a result. The calculated accuracy value is shown within the place where the braces indicate a filtered result usually not shown to the reverse engineer.

In the first case, where the reverse engineer accepts the filtered result, we store 40% as virtual threshold in the set  $th_v(b:Bridge)$ . Note, in comparison to the previous described interaction of the reverse engineer, he/she only accepts or rejects the result. The accuracy value has not changed.

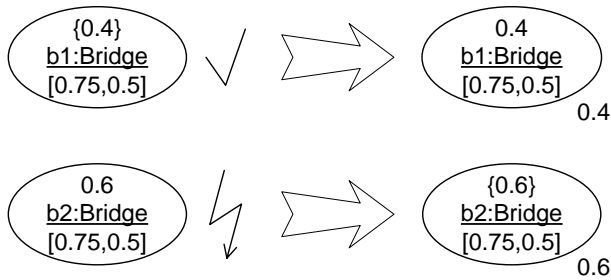


Figure 11: Correction of the threshold

The result will only be visible or not, respectively. Therefore the calculation of the  $cac$  value is not touched and thus, no other virtual thresholds are created. The same applies to the second case, where the reverse engineer rejects a result. In this case we store 60% as virtual threshold.

### 4.2.3 Average Calculation

While the reverse engineer has corrected some accuracy values and accepted and rejected results, a number of virtual accuracy values and virtual thresholds have been created and assigned to the places. Now, we take these virtual values to adapt the credibility value and the threshold of the pattern rules. Since both adaptations are equal, in the following we only describe the calculation of the new credibility value of a rule. The same holds for the new threshold of a pattern rule.

DEFINITION 4. Let for a place  $p \in P$  be  $n_p = |ac_v(p)|$  the number of virtual accuracy values of place  $p$ , then the average virtual accuracy value of a place is defined as  $\overline{ac}_v : P \rightarrow [0..1]$  with  $\forall p$  in  $P$  is

$$\overline{ac}_v(p) = \begin{cases} \frac{1}{n_p} \sum_1^{n_p} ac_v(p): & \text{if } n_p \neq 0 \\ ac(p): & \text{else} \end{cases}$$

In the first step, we calculate the average virtual accuracy value for each place in the FPN, cf. Definition 4. If the reverse engineer has not corrected the calculated accuracy value, we take this value as virtual average. This first average calculation eliminates values that cancel each other out<sup>2</sup>. In the second step we use all virtual values to calculate the new credibility value for a rule. Therefore we distinguish between two variants.

DEFINITION 5. Let  $place_r$  be the set of all places corresponding to annotations created by pattern rule  $r$ . Then we define as first variant of the new credibility value  $cr$  of a rule  $r$  as

$$cr'_r = \frac{1}{1 + |place_r|} [cr_r + \sum_{p \in place_r} \overline{ac}_v(p)]$$

The first adaption variant shown in Definition 5 takes all annotations respectively places into account that have been created by a certain rule  $r$  and assigns the new credibility value  $cr'_r$  as average over all these places to the rule. This has the effect, that in situations where a large number of annotations exist and the reverse engineer has corrected only

<sup>2</sup>Currently, we have implemented the average calculation directly, whenever the reverse engineer performs changes. Therefore we loose the history information.

some of them the regression is very low. If the reverse engineer wants that his/her changes result in a higher regression, he/she can use the adaption in Definition 6.

DEFINITION 6. Let  $corplace_r$  be the set of all places corresponding to annotations created by pattern rule  $r$  where  $\overline{ac}_v(p) \neq ac(p)$ . Then we define as second variant of the new credibility value  $cr$  of a rule  $r$  as

$$cr''_r = \frac{1}{1 + |corplace_r|} [cr_r + \sum_{p \in corplace_r} \overline{ac}_v(p)]$$

Definition 6 takes only those places into account, which have an average virtual accuracy value different from the calculated one. These are the places where the reverse engineer has corrected the accuracy value and all preceding places where the accuracy value has been modified according to 4.2.1.

Note, the reverse engineer can decide which kind of adaption he/she wants to use. Our prototype allows the reverse engineer to choose the kind of adaption. It is important that the reverse engineer is responsible for the kind of adaption and the time when the adaption takes place, i.e., the user has to start the adaption, explicitly.

In addition, the reverse engineer is also able to reset the virtual values. This action allows for eliminating wrong corrections of the reverse engineer as well as for skipping corrections not to take into account for adaption. Thus, it is different from an undo of the correction, because the modifications and a recalculation is left untouched.

## 5. Conclusions and Future Work

In [20] we presented our graph-grammar-based pattern definition language and a semi-automatic inference algorithm. This algorithm provides meaningful results early, especially in large software systems. It therefore enables a reverse engineering process where the reverse engineer can iteratively test and adapt a given pattern rule catalog to the system under investigation.

In order to assist the reverse engineer to assess the results of the pattern inference, we introduced credibility values for pattern rules [21]. Annotations created by the pattern rules get accuracy values, see 4.1, calculated from the credibility values. The credibility values are initially estimated by the reverse engineer.

In this paper we present an approach to adapt the credibility values within the iterations of the analysis process. Since the basis for the adaption, the pattern rule catalog, changes during the iterations, classical learning approaches such as fuzzy neural nets are not suitable. We use simple statistical analyses to adapt the credibility values of the rules. During the iterations the reverse engineer evaluates and accepts or rejects the results. These changes are the input for the adaption of the credibility values and thresholds.

To evaluate our prototype, we analyzed Java's abstract window toolkit (AWT) library, the Jigsaw web server and the Fujaba Tool Suite environment, wherein the prototype is implemented as a plug-in. During our tests we observed, that our adaption approach is reliable to determine useless pattern rules. If the credibility value of a pattern rule falls below a certain value, the pattern rule may not be reliable. For example a credibility value of only 20% indicates that the rule produces too much false-positives. Therefore the

pattern rule is counterproductive, i.e., the structure of the rule should be changed.

In cooperation with other universities and industrial partners we will evaluate our reverse engineering prototype and process in a project environment in the near future. During the evaluation we want to discuss open questions and collect experience for the question which annotations should be used for the adaption and therewith which kind of calculation is appropriate for a certain case or certain system, cf. Definition 5 and Definition 6. Another question is the independent adaption of the credibility value and the threshold. The reverse engineer uses two different methods to adapt both values. We expect that a method which adapts the credibility value and the threshold in one step results in a better regression.

Our reverse engineering plans are that we enhance our approach with dynamic analysis techniques. Static analysis indicates pattern candidates and a dynamic analysis verifies their existence. The candidates are also used to reduce the search space of the dynamic analysis. The basic ideas are described in [26].

Another future perspective we currently focus on is the detection of Anti Patterns and to investigate their occurrences in comparison to normal patterns. This results from an observation we made analyzing an industrial system, where the initial design includes patterns but during system maintenance this clear design has been ignored and partially replaced by 'hacking' solutions. Our idea is to assign additional property values to the rules and also to the annotation results. Based on the property values we want to re-factor 'bad' patterns by 'good' patterns like the ideas presented in [13].

## 6. References

- [1] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
- [2] M. Brown and C. Harris. *Neurofuzzy Adaptive Modeling and Control*. Prentice Hall, 1994.
- [3] F. Buschmann, R. Meunier, H. Rohnert, P. Somerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, Inc., 1<sup>st</sup> edition, 1996.
- [4] V. Cherkassky and F. Mulier. *Learning From Data*. John Wiley and Sons, Inc., 1998.
- [5] J. Coplien, N. Kerth, and J. Vlissides. *Pattern Languages of Program Design (Volume 3)*. Addison-Wesley, 1996.
- [6] J. Ebert, R. Gimnich, and A. Winter. *Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO*, pages 263–275. Gabler, 1996.
- [7] A. Fay and E. Schnieder. Fuzzy petri nets for knowledge representation and reasoning in rule-based systems. In N. Steele, editor, *Proceedings of the 2<sup>nd</sup> International ICSC Symposium on Fuzzy Logic and Applications, Zurich*, pages 146–150. ICSC Academic Press, 1997.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [9] M. T. Harandi and J. Q. Ning. Knowledge based program analysis. *IEEE Transactions on Software Engineering*, 7(1):74–81, 1990.
- [10] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *Proc. of the 11<sup>th</sup> International Workshop on Program Comprehension (IWPC), Portland, USA*, May 2003.
- [11] J. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.
- [12] J. Jahnke and C. Strebin. Adaptive tool support for database reverse engineering. In *Proc. of 1999 Conference of the North American Fuzzy Information Processing Society, New York, USA*, June 1999.
- [13] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. In S. Demeyer and H. Gall, editors, *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical Report TUV-1841-97-10, Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997.
- [14] A. Konar and A. K. Mandal. Uncertainty management in expert systems using fuzzy petri nets. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):96–105, February 1996.
- [15] B. Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Approach to Machine Intelligence*. Prentice Hall, 1992.
- [16] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.
- [17] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1<sup>st</sup> edition, 1984.
- [18] H. Müller and K. Klashinsky. Rigi - a system for programming-in-the-large. In *Proceedings of the 10<sup>th</sup> International Conference on Software Engineering (ICSE 1988), Singapore*, pages 80–86. IEEE Computer Society Press, 1988.
- [19] J. Niere. *Incremental Design-Pattern Recognition*. PhD thesis, University of Paderborn, Paderborn, Germany, 2004. to appear.
- [20] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24<sup>th</sup> International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.
- [21] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the 11<sup>th</sup> International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press, May 2003.
- [22] Y. Pao. *Adaptive Pattern Recognition and Neural Networks*. Addison-Wesley, 1989.
- [23] A. Radermacher. Support for design patterns through graph transformation tools. In *Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands, LNCS 1779*. Springer Verlag, 1999.
- [24] B. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

- [25] SUN Microsystems, Inc. *JavaBeans API Specification 1.01*, Online at <http://java.sun.com/products/javabeans/docs> (last visited June 2003), 1997.
- [26] 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.
- [27] P. Werbos. *The Roots of Backpropagation: From Ordered Derivations to Neural Networks and Political Forecasting*. John Wiley and Sons, Inc., 1994.
- [28] L. Wills. *Automated program recognition by graph parsing*. PhD thesis, MIT, Cambridge, Mass., 1992.
- [29] L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.
- [30] L. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1978.
- [31] A. Zündorf. Graph pattern matching in progres. In *Proc. of the 5<sup>th</sup> International Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 1073. Springer Verlag, 1996.