Recovering UML Diagrams from Java Code using Patterns

Jörg Niere
Department of Mathematics
and Computer Science
University of Paderborn
Warburgerstraße 100
33098 Paderborn, Germany
nierej@uni-paderborn.de

Jörg P. Wadsack
Department of Mathematics
and Computer Science
University of Paderborn
Warburgerstraße 100
33098 Paderborn, Germany
maroc@uni-paderborn.de

Albert Zündorf
Department of Mathematics
and Computer Science
University of Paderborn
Warburgerstraße 100
33098 Paderborn, Germany
zuendorf@uni-paderborn.de

ABSTRACT
Recovering the static structure of legacy source code e.g. as an UML class diagram is quite well understood. In contrast, recovering high-level behaviour diagrams from source code is still an open issue. This paper proposes to use fuzzy pattern detection techniques for the recovery of UML collaboration diagrams from source code. The approach is based on a knowledge base of basic datatypes and of generic collection classes and of code clichés for Java beans and of fuzzy patterns for object structure look-up and modification clichés. We handle the diversity of existing code clichés by organizing them in an object-oriented hierarchy factorizing important common properties and by relaxing exactness requirements for cliché detection with the help of fuzzy theory. We handle the runtime efforts for cliché detection using a sophisticated inference mechanism based on generic fuzzy reasoning nets (GFRN's). The work is part of the FUJABA case tool aiming to support round-trip engineering for UML and Java.

Keywords
UML, Java, fuzzy logic, reverse-engineering, pattern recognition, round-trip engineering

1 INTRODUCTION

Reverse engineering aims to provide program descriptions on higher levels of abstractions. Such an abstract level could e.g. be a program description using UML diagrams. These program descriptions facilitate the understanding of program structures and program behaviour. State-of-the-art CASE tools like Rational Rose [Ros], TogetherJ [Tog], and Rhapsody [Rha] provide only recovery functions for class diagrams using markers in the code. Notably, Rhapsody supports the recovery of state-charts. The recovery of high-level behaviour descriptions for legacy object-oriented programs is still an open issue.

The work described in this paper is part of the Fujaba project. The Fujaba project aims to develop a round-trip engineering CASE tool for UML. In our previous work, [FNTZ98, JZ98, NNSZ99, KNNZ00], we proposed an execution semantics for UML statecharts, activity diagrams, and collaboration diagrams. Our execution semantics allows to use these UML behaviour diagrams as a visual programming language for object-oriented applications. The code generators of the Fujaba environment translate such executable specifications into fully functional Java classes including method bodies.

Theoretically, with Fujaba no manual coding is necessary any more. Practically, the generated code is frequently modified during debugging. The code may be merged with the contributions of other developers e.g. via a configuration management system. Some system parts may be added by other code generators, e.g. a GUI builder or a database middleware layer, or a distribution layer like CORBA [Vin97].

To deal with such code modifications, the Fujaba environment provides reverse engineering support that analyses Java source code and tries to create the corresponding UML class and behaviour diagrams, cf. [NNWZ00]. So far, the reverse engineering capabilities of Fujaba are limited to round-trip engineering support. This means, Fujaba is merely able to reverse engineer code it has generated itself or that it has been written as if it would have been generated.

Reverse engineering of arbitrary legacy code and third party code is a challenging problem. Most CASE tools are restricted to the analysis of static program structures, i.e. reverse engineering of class diagrams. But even for class diagrams the correct recovery of (structural) associations between classes is not trivial.

To overcome these limitations and to be able to deal with legacy code constructs, this paper proposes the use of fuzzy reasoning technologies, i.e. generic fuzzy reasoning nets, cf. [Jah99]. In this paper we focus on the analysis of code that deals with object structure modifications. We start with a knowledge base about the semantics of pre-defined container classes and their access operations. We analyse attribute declarations to identify (sets of) basic references hold by certain classes. Methods modifying these basic references are classified as access methods with certain degrees of confidence. If access methods are detected with sufficient confidence, their use can be analysed. Specific sequences of access method usages may be turned into collaboration diagrams that describe the look-up of certain object patterns and the modifications of such patterns and the collaboration messages send between the participating objects. The

combination of activity diagrams specifying the control flow and embedded collaboration diagrams in a certain activity is called story-diagrams. Those story-diagrams serve as the behavioural specification for a software system and are linked to method declarations in class diagrams.

The following Section 2 introduces a track based material transportation system as running example. In Section 3 the reconstruction of class diagrams and story-diagrams is described using annotations. The specification of code clichés is introduced in Section 4 and the following Section 5 introduces the corresponding execution formalism, namely generic fuzzy reasoning nets. Section 6 and Section 7 discuss related work and present future work.

2 RUNNING EXAMPLE: SWITCH CONTROL SOFTWARE

In this section, we introduce the switch control software of a track based material transportation system as running example. This example stems from the joint research project ISILEIT funded by the german research foundation (DFG). Within ISILEIT we collaborate with our mechanical and electrical engineering department to set-up an agent based production control system. The building blocks of such a production control system are different, self-acting and computer controlled resources like e.g. switches, shuttles, machines, or robots. Shuttles move on rails and transport goods between various production places. Each production place can be reached using switches in the railway system. Shuttles announce themselves at the switches if they want to visit the corresponding production place. The switch control software keeps track of the targets of the different shuttles and operates the switch accordingly.

Figure 1. shows the structure of a switch as part of a production control system, which we specified by employing Fujaba. The switch has an actor, i.e. the switch drive, which changes its direction. Further it has some sensors, which observe the environment and a Local Operating Network-node, which is connected to a communication network. In our example, the identification unit detects an arriving shuttle and reports the shuttle's id to the switch control node. Now, the control software decides in which direction the shuttle should be send. If the switch has to change its direction, it activates the stopper in order to let shuttles wait. One has to assure, that no shuttle is in the switching area, when the switch drive is activated, because otherwise the switch drive could be damaged. For that reason, the switch has a pass observer at each exit, which reports every shuttle leaving the area. Note, that we have a one-way driving direction, so that we have one entry and two exits, which means that our example shows a 'branching switch '.

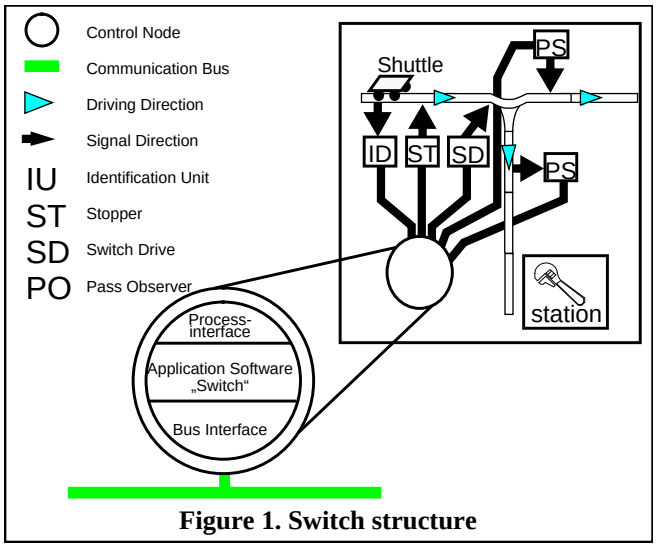

Figure 1. Switch structure

We have developed the control software for such switches using our Fujaba environment. Then, we delivered our software to the mechanical engineers setting up the physical transportation system. Some weeks later we faced the situation, that the mechanical engineers modified the software significantly for debugging and optimization reasons. Thus, we wanted to be able to reconstruct UML class and behaviour diagrams from the changed Java code that reflect the current control software.

3 CLASS- AND STORY-DIAGRAM RECONSTRUCTION

Recovering class and behaviour diagrams from Java code is divided into two tasks. First, the static information, the class diagrams, will be reconstructed and in a second task, the behaviour diagrams (here story-diagrams) will be recognized.

```
 1: public class Switch extends Track {
 2:    ...
 3:    int shuttleId;
 4:    ...
 5:    Stopper stopper = new Stopper ();
 6:    ...
 7:    OrderedSet announced = new OrderedSet ();
 8:    ...
 9:    void welcome (int id) {...}
10:    ...
11: } // class Switch
```

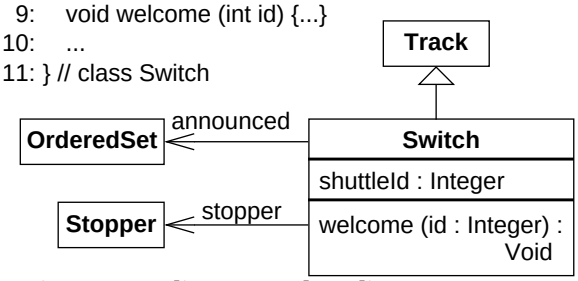

Figure 2. Rudimentary class diagram recovery

Figure 2. shows a cut-out of the static elements of the Java code of class `Switch`. From this code fragments a rudimentary class diagram recovery approach could reconstruct the class diagram shown below the Java code. Classes become classes. Attributes of basic types like `int` or `boolean` become class attributes. Method declarations become methods of the corresponding UML classes. Inheritance in Java is directly mapped to inheritance relations in the diagram.

We assume, that the class diagram recovery mechanism has already knowledge about all basic Java types. Thus, it may identify types Stopper and OrderedSet as user defined types. Accordingly, the corresponding Java attributes are interpreted as references in the class diagram.

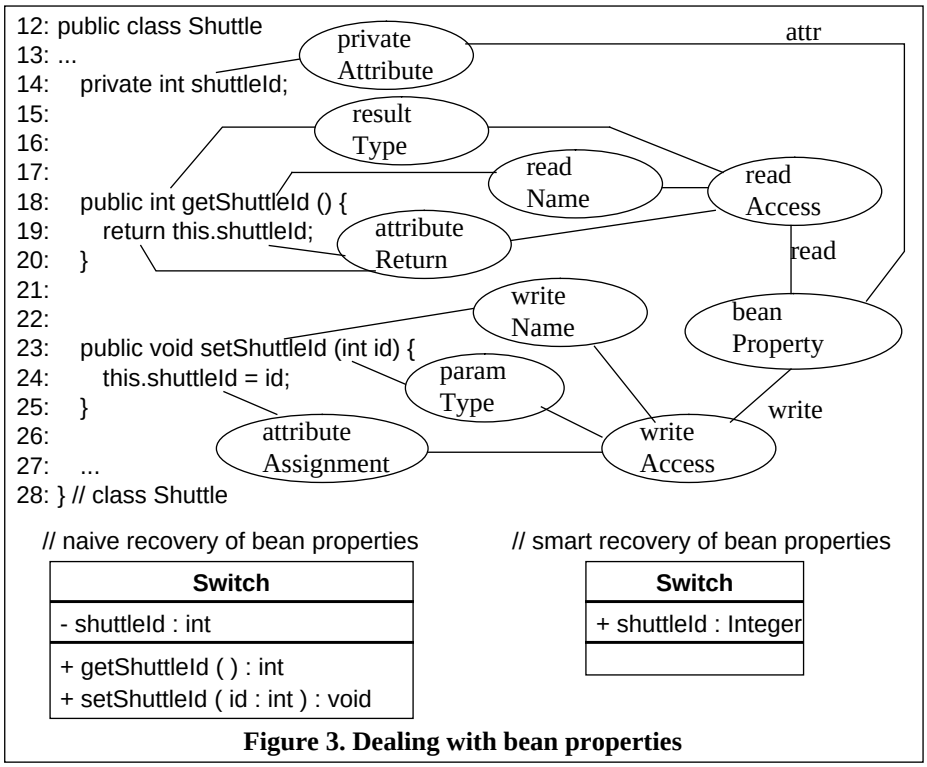

Figure 3. Dealing with bean properties

However, class OrderedSet is a pre-defined generic container class from the Java Foundation Class (JFC) library. Equipping our reconstruction mechanism with this additional knowledge, it could turn the announced reference from Switch to OrderedSet into a to-many reference to class Object (the basic class of all classes in Java). In Java, we face the problem that generic container classes do not provide information about the types of the contained entities. To derive such information, our recovery mechanism needs to know the semantics of the access methods of container classes, e.g. method add inserts elements into the container. This allows us to derive the entry type for containers from the usages of the corresponding add method.

Recognizing classes and class members is fairly simple. Analysing method bodies is a more challenging task. One approach to recover the semantics of method bodies is the detection of so-called code clichés, cf. [Wil94]. In Java, very common code clichés are bean properties. A bean property is an attribute with appropriate read and write access methods, cf. Figure 3. The read (write) method of a bean property must have the same name as the corresponding attribute plus a 'get' ('set') prefix. In addition, the write method must have exactly one parameter with the same type as the corresponding attribute. Finally, within the body of the write method the parameter value must be assigned to the corresponding attribute. Once an attribute and its access methods have been classified as a bean property, the class diagram recovery mechanism may simplify the corresponding class, accordingly.

In our approach, we use similar code clichés to implement bi-directional associations. Bi-directional associations are implemented using pairs of pointers. These pointers are encapsulated with appropriate read and write access methods. The write access methods guarantee the consistency of the pointer pairs by calling each other, mutually. For to-one associations simple attributes and set- and get-methods are used. For to-many associations we employ generic container classes and methods for iterating through the set of neighbours, adding neighbours, and removing neighbours.

Fujaba employs a flexible cliché detection mechanism the so-called annotation engines. The annotation engines enrich the abstract syntax tree of a parsed program with so-called annotations [HN90]. Annotations are markers for detected occurrences of code clichés. In Figure 3 such annotations are shown as ovals. Annotations enrich the semantics information of abstract syntax trees and allow e.g. to simplify class diagrams. Moreover, such annotations assign a certain semantics to certain methods or code fragments. This semantics may be used for further analysis of other method bodies.

Consider for example Figure 4. Line 35 employs method `getIdUnit`. Let us assume that method `getIdUnit` has been annotated as the read access method for an association between class `Switch` and class `IdentificationUnit`. This allows us to interpret line 35 as a link look-up operation. In a collaboration diagram such a link look-up operation is shown as a line labelled with the corresponding association name. Such a line connects two boxes representing the source and target variable. In our example these are the variables this and idU, respectively, cf. Figure 4. Similarly, the knowledge about access methods

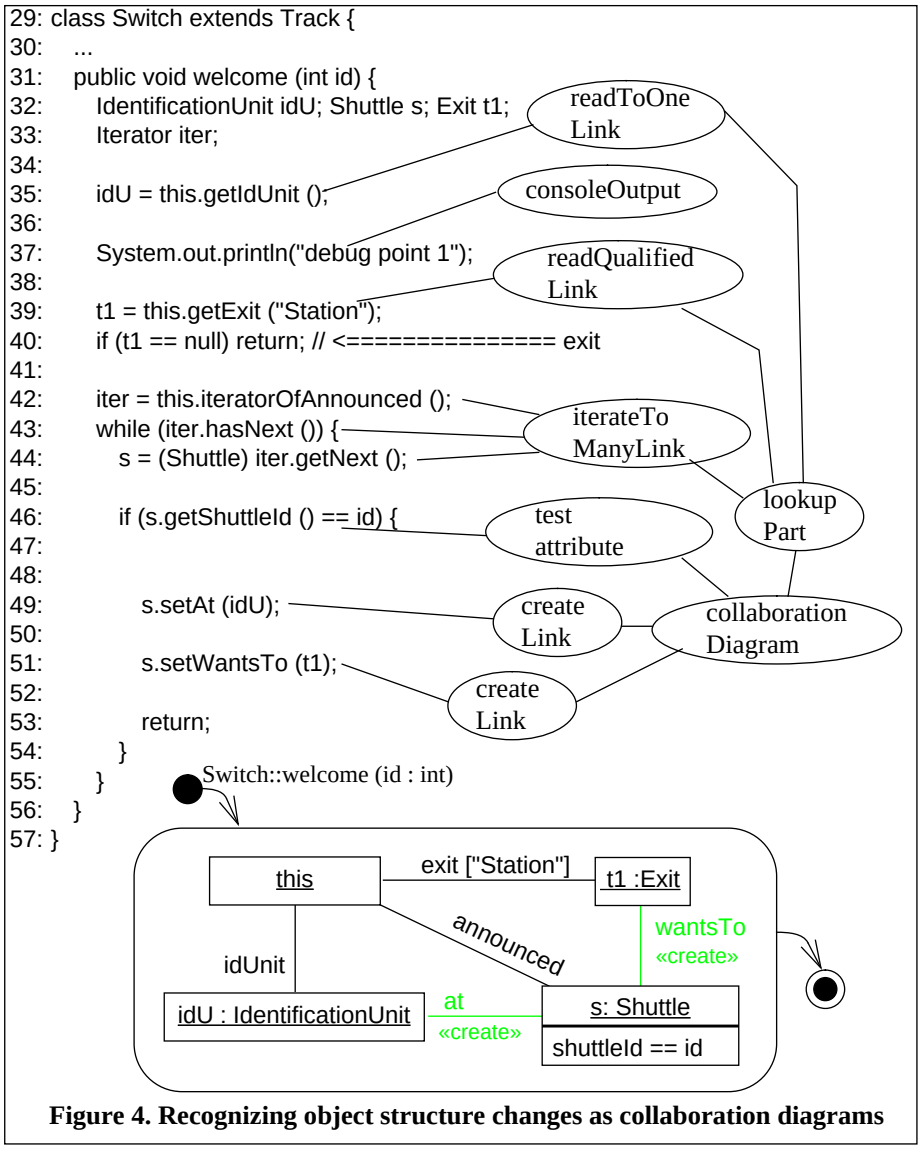

Figure 4. Recognizing object structure changes as collaboration diagrams

may allow to interpret line 39 and 40 as look-up of a qualified association with cardinality 0..1. Lines 42 to 44 show a typical cliché for the look-up of a to-many association. In a collaboration diagram such look-ups are shown as lines between appropriate objects, cf. the bottom of Figure 4. Finally, our annotation knowledge allows us to infer, that line 49 creates an at link between object s and idU and line 51 creates a wantsTo link from s to t1. In the collaboration diagram we show link creation using grey colour and the «create» stereotype.

Thus, the detection of Java bean property clichés allows us to assign a dedicated semantics to read- and write-access methods. This knowledge allows us to analyse the usage of such access methods and to recover collaboration diagrams from such code. Note, in more complex situations, a method body may contain several code fragments that correspond to collaboration diagrams. Such code fragments may be mixed with other code (Figure 4 line 37) and control structures that are not covered by such an analysis. To deal with such situations, we embed detected collaboration diagrams into activity diagrams. The activity diagram part shows the top-level control flow and text activities for unrecognized code. Recognized code is turned into collaboration diagram activities.

So far, our annotation engines are able to deal with code that strictly conforms to the Fujaba code generation concepts. Due to our experience, legacy and third party code frequently contains similar code fragments. However, the classification of access methods and especially the detection of collaboration diagram operations is very challenging. For example, there are numerous ways to implement a to-many association. Accordingly, there are very different ways to enumerate all neighbours of a given object. Similarly, there are various coding clichés for test operations and for dealing with test results. The next section describes how clichés can be specified using the Fujaba environment.

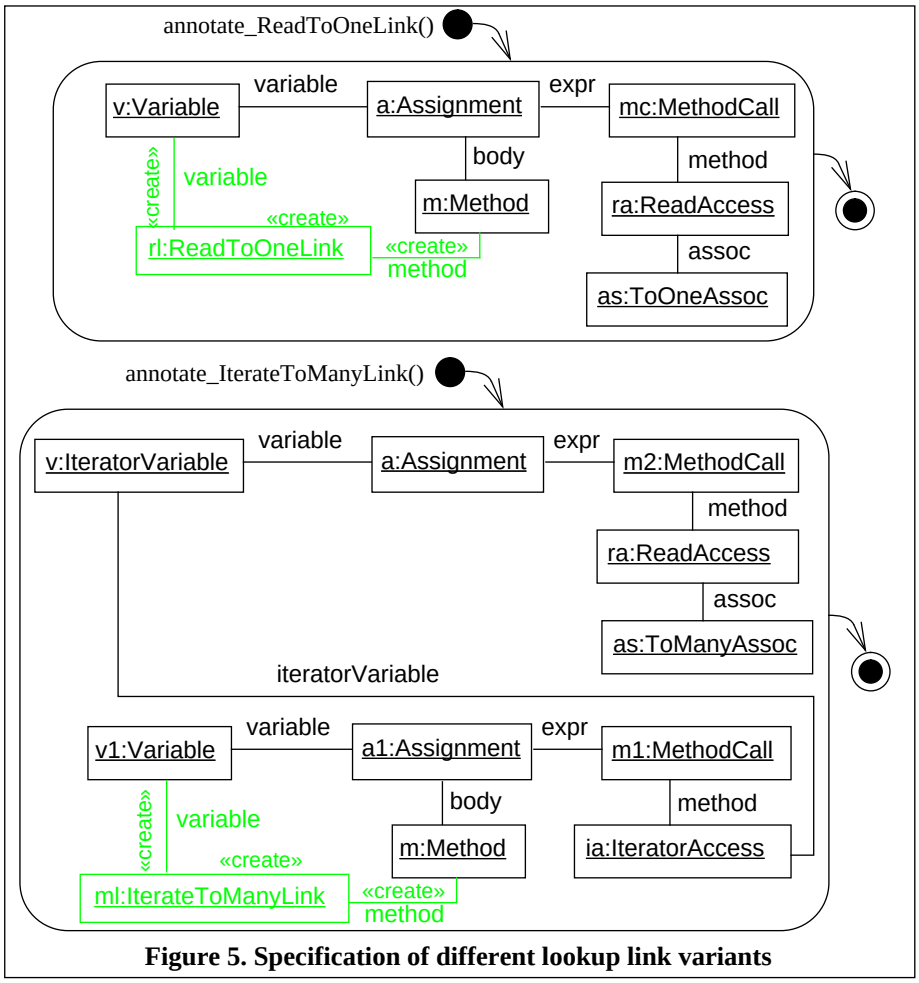

Figure 5. Specification of different lookup link variants

4 ANNOTATION-ENGINE SPECIFICATION

In general, one annotation engine is responsible for the detection of one cliché. The annotation engines work on an abstract syntax graph constructed out of the code (As parser we use JavaCC [JCC]). Each annotation engine is embedded in an architecture which is mainly a combination of a strategy and a chain of responsibility pattern [GHJV95]. Changed objects of the syntax graph are passed to a broadcaster which distributes the objects to those engines registered for the objects' type. The main part to specify a code cliché relies on a method implementation for the corresponding annotation engine. The annotation methods are illustrated using activity diagrams and collaboration diagrams i.e. Fujaba story diagrams. Figure 5. shows collaboration diagrams for the detection of link annotations. The upper part of Figure 5. shows an activity diagram for the annotation of a readToOneLink (cf. Figure 4 line 35). If an assignment in a method body consists of a variable and a method call, which is annotated as read access of a 'to-one' association, then we annotate the variable and the method with an ReadToOneLink node. The annotation of a readQualifiedLink (cf. Figure 4 line 39) is similar to the readToOneLink annotation. Only node as:ToOneAssoc in the collaboration diagram on the top of Figure 5. has to be replaced by as:ToQualifiedAssoc. Also the created annotation rl:ReadToOneLink has to be replaced by a variable of type ReadQualifiedLink. For the annotation of the more complex IteratedToManyLink story-diagram we have to check the read access over an iterator construct, cf. bottom of Figure 5.. Basically, a first assignment has to retrieve an iterator and a second assignment must use this iterator to assign a value to the actual target variable. Therefore, the lower part of the annotate_IterateToManyLink collaboration diagram refers to line 44 and the upper part matches line 42.

To get an overview of the detectable code clichés, we developed a domain model for the annotations of code patterns. Figure 6. shows a cut out of the domain model (as an UML class diagram) used for the round-trip facilities supported by the Fujaba environment. This model shows the annotations for the class diagram on the left and the story diagram annotation parts on the right.

Using inheritance hierarchies allow some simplification in the specification of the annotation engines. Without the inheritance hierarchy collecting all look-up links of a method and aggregate them with a LookupPart annotation, may result in three annotation methods. I.e. three annotation engines, one collecting ReadToOneLinks, another ReadQualifiedLinks and an engine collecting ReadToManyLinks.

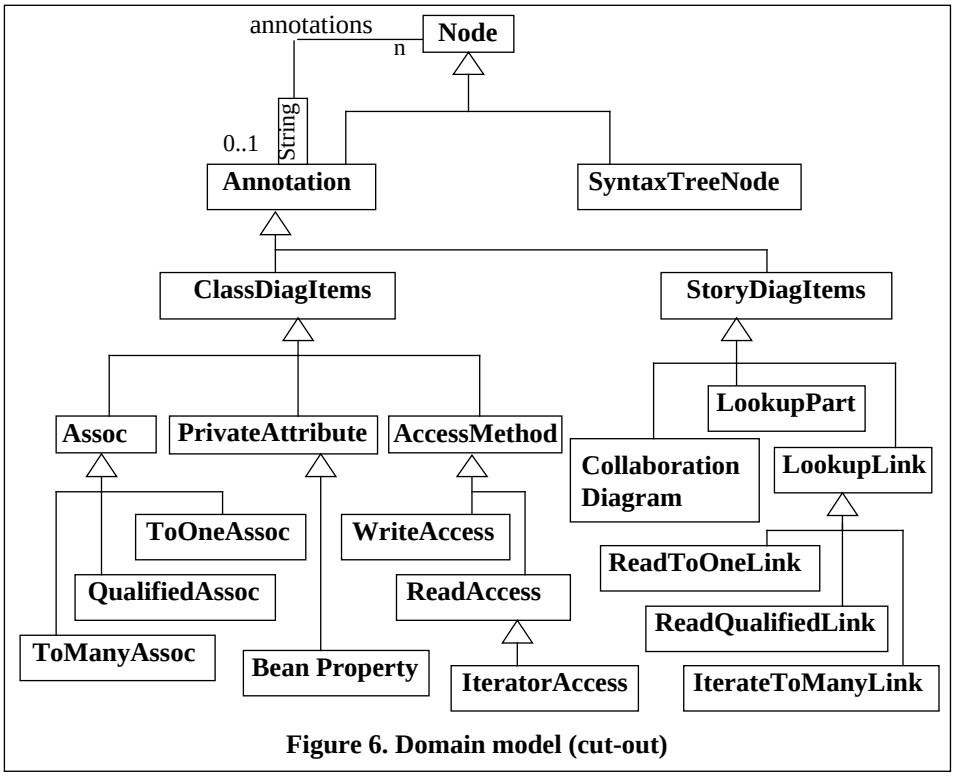

Figure 6. Domain model (cut-out)

Instead we can use one method (engine) using the superclass LookupLink of the three read link classes. This reduces the number of required methods. Figure 7. shows the annotation for LookupPart containing a set of LookupLink nodes (depicted by two stacked boxes), e.g. ReadToOneLink, ReadQualifiedLink and IterateToManyLink (cf. Figure 4). In general in a collaboration diagram must not contain additional lines like debug outputs (cf. Figure 4 line 37). Such negative application conditions may be specified by negative (crossed-out) nodes. Therefore, the crossed-out node co:ConsoleOutput in Figure 7. assures that a lookup part is only annotated if there is no output printed on the console. For patterns on a high level of abstraction the

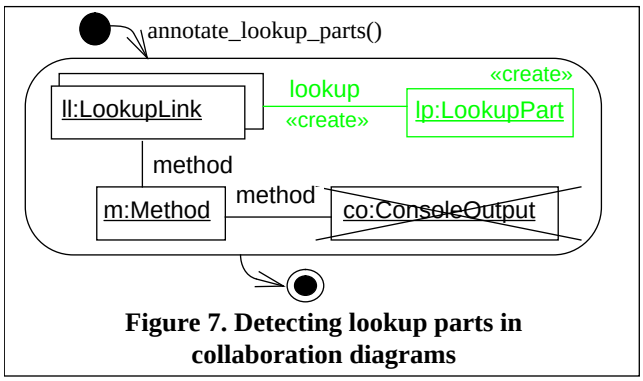

Figure 7. Detecting lookup parts in collaboration diagrams

specification using pure collaboration diagrams is sufficient enough. In general, every time a boolean answer is expected and easily given, collaboration diagrams are sufficient. Looking for clichés (code pattern) is a little bit more difficult, because there are many more variants (e.g. syntax) to express the same semantics. For example, some developers prefer while-loops and other for-loops to solve the same problem.

Such variants highly depend on the education and social background or on affectations of a specific developer. Using collaboration diagrams to specify clichés lead more or less to one rule for one cliché. To deal with this problem, we relax the exactness of cliché detection using fuzzy logic. This allows us to downsize the cliché detection of certain indicators that signal the existence or absence of certain clichés with a certain confidence. For example in Figure 4 the three variants (line 35, line 39 and line 44) of binding an object to a variable annotated with a subclass of LookupLink results in three completely different specifications if we are using collaboration diagrams, only (cf. Figure 5).

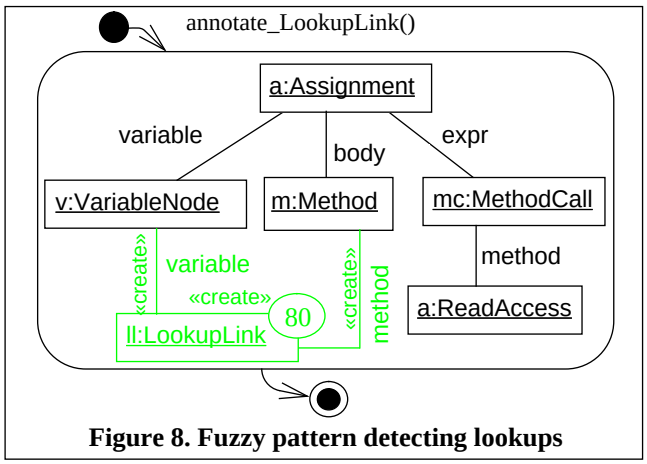

Figure 8. Fuzzy pattern detecting lookups

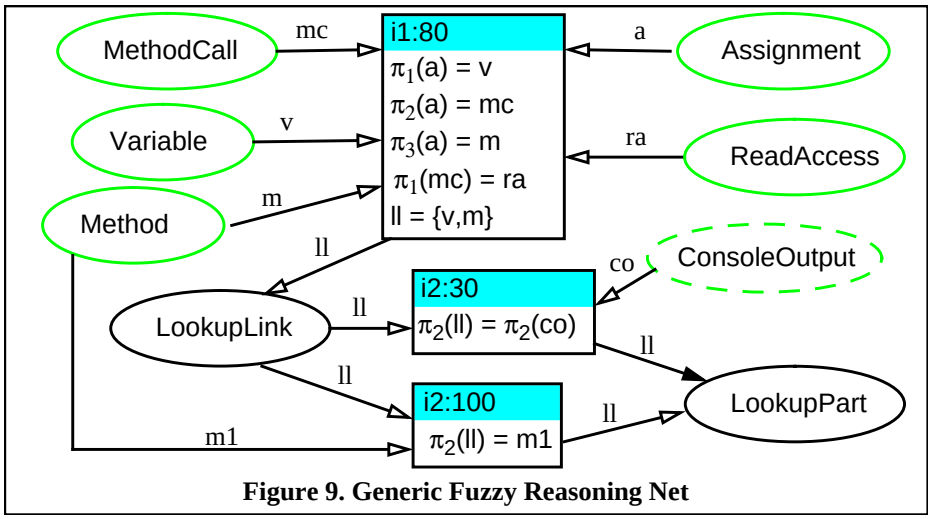

Figure 9. Generic Fuzzy Reasoning Net

This results from the fact, that the abstract syntax trees for the three assignments is different. However, all three rules employ the same indicator, the usage of an association look-up method and an assignment. Thus, we may use only a single rule detecting the usage of a read access method within an assignment, cf. Figure 8.

Also the three rules have many structural similarities, which makes them gluing candidates. ReadToOneLink and ReadQualifiedLink only differ in the type of the variable as. ReadToManyLink seems not to be so close to the other two, e.g. there are more variables used. But, taking the inheritance hierarchy into account and put the common parts into one collaboration diagram forces uncertainty in the detection. To handle the uncertainty coming up with the abstraction of the three collaboration diagrams, we indicate the annotation with a confidence (fuzzy-value) of 80. Collaboration diagrams added with fuzzy-values, are called *fuzzy patterns*. Figure 8. shows the resulting fuzzy pattern for general lookup-links, which is a combination of the three variants ReadToOneLink, ReadQualifiedLink and ReadToManyLink before. The fuzzy value of the pattern is put in a circle at the top right corner of the variable, here the new annotation. A fuzzy pattern fires if its premise is fulfilled which means in this case, that all variables specified in the pattern must be bound to an object of the abstract syntax graph with a fuzzy value that is higher than the required value. The result value for created annotations is the fuzzy-and of the fuzzy beliefs of the found objects and the fuzzy value of the created object.

If the pattern could be matched, the resulting annotation which is created, gets as fuzzy value positive 80. This reflects that due to our experience such a match is in 80 percent of all cases a correct match and the underlying source code refers to a binding object situation. Having a deeper look in the source code, we find that the fuzzy pattern matches for line 35, line 39, and line 42 instead of line 44. Line 42 is not exactly the line where the object is bound, this is done in line 44, but such uncertainty could be expressed through fuzzy-values and enables us to reduce the number of patterns in the reengineering process.

In those cases, where it is necessary to decide whether or not the automatic decision is correct, the reengineer has to be asked. Typically, the reengineer has to look for nodes with a fuzzy value minor than a defined limit and could overwrite the value to zero or 100. The changes trigger a re-evaluation of the fuzzy values of the depending nodes. Experiences have shown that such a semi-automatic approach causes better results than fully automatic approaches, because a user or reengineer may interact with the tool and contribute her/ his knowledge.

5 DETECTION MECHANISM

For detecting clichés in source code we use the formalism of *Generic Fuzzy Reasoning Nets* (GFRN) [JSZ97, Jah99]. The GFRN formalism has initially been applied in the domain of data reverse engineering. It facilitates the specification and execution of analysis rules and processes and incorporates a notion of uncertainty. In principle, a GFRN is a net of *predicates* (oval shape) and *implications* (represented as boxes) which are connected by arcs (cf. Figure 9). Arcs are labelled with formal parameters that can be used to specify constraints for implications. Negation in implications are represented by arcs with black arrow heads. Each implication has an associated *confidence value* (CV). Based on the theory of possibilistic logic [DLP94], the semantics of a CV is a lower bound of the necessity that the corresponding implication is valid. Note, the CV associated to the implications specify a measure for the degree of certainty. According to typical fuzzy inference operators, an overall valuation for each cliché is based on the difference between the maximum positive CV and the maximum negative CV.

Each specified fuzzy pattern, is canonically mapped to a (part) GFRN. Afterwards, the (part) GFRN's are merged, i.e. predicates with equal names are glued, resulting in one or more larger nets. Figure 9 shows the GFRN after merging the canonically mapped fuzzy pattern shown in Figure 8 and a fuzzy pattern constructed out of Figure 7.. Variables that occur in the fuzzy pattern are mapped to predicates and are rendered in grey colour while the annotation which represents the goal (created annotations) of the analysis is mapped to a so-called *dependent* predicate LookupLink,

respectively LookupPart, rendered in black. The relations between variables, i.e. links between variables are mapped to constraints within the corresponding implication ($\pi_1(a) = v$). Therefore, the data represented by the variable in a fuzzy pattern is mapped to formal parameters in the GFRN. For example, variable a:Assignment is mapped to an Assignment predicate and the data, i.e. the variable, the method, the expression and also the assignment itself, is mapped to the formal set parameter a at the transition from predicate Assignment to implication i1.

For efficient fuzzy pattern analysis, we propose a two-step process. First, the fragment graph is searched for a match for all positive variables. Subsequently, our detection strategy aims to extend each such match by a match for negative (cancelled) nodes in a fuzzy pattern. The GFRN formalism facilitates the specification of such a strategy by distinguishing between so-called *data-driven* and *goal-driven* predicates. Matches for data-driven predicates (represented with solid grey outline) are searched at the beginning of the analysis process (cf. MethodCall, or and Variable in Figure 9). Subsequently, the fragment graph is searched for matching instances of goal-driven predicates (with dashed grey outline). For example, the ConsoleOutput predicate is mapped to a goal-driven predicate and allows us to specify that a console output results in a negative belief of 30 if it occurs within some lookup links of a method. This is different to Figure 7., where a console output must not appear between two lookup-links of a method. This can also be expressed in a corresponding fuzzy pattern (not shown here) and helps us to reduce the number of fuzzy patterns for code clichés. We refer to [JH98] for details on the GFRN inference engine.

6 RELATED WORK

The FUJABA environment and especially its code generators are described in detail in [FNTZ98, KNNZ00]. The underlying technique called story-driven-modelling starting from the first phases of a software development process in general is described in [JZ98] and specializations for production control systems are presented in [NNSZ99].

[HN90] proposes a program analysis based on an *Event Base* and a *Plan Base*. First, rudimentary events are constructed from source code. Plans are used to consume one or more events and fire a new event which correspond to the plan's intention. Annotations visualize the event flow and plan definition. In [PP94] a matching algorithm for syntactic patterns based on a non-deterministic finite automaton is presented. The non-determinism is used to provide dummy variables for special pattern symbols representing syntactical information like variables or function calls. Both [HN90] and [PP94] need one definition for one implementation variant, which lets the approaches fail for at least legacy systems with unknown code-styles.

An automatic approach to extract semantics information from source code is presented by Wills [Wil94]. Similar to our collaboration diagrams, Wills uses graph rewrite rules to specify implementation patterns in terms of so-called *program plans*. Program plans are structured in a hierarchical design library, i.e., abstract plans consist of aggregations of several more simple plans. A specific library stores program-plans for different domains, e.g., Wills presents a library for sorting algorithms. Wills follows a bottom-up strategy to detect plans which limits the practical usage of the approach to source code about 1000 lines. Quilici [Qui94] proposes an indexing technique and combines top-down and bottom-up detection to overcome this problem.

In [KSRP99], Keller et al. present a semi-automatic approach to find design patterns [GHJV95] in source code. Patterns are represented in UML notation [BRJ99], namely in CDIF format. Matching algorithms are not automatically generated but implemented by hand. [Rad99] employs graph transformations (graph rewrite rules) to extract design patterns automatically and to refacture parts of the source code. Graph rewrite rules are similar to our collaboration diagrams without fuzzyness, so the approach is restricted to the restrictions discussed in section 4.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces the round-trip facilities of the Fujaba environment based on a material flow system specification. The control software for the material flow system is generated out of a Fujaba specification, adapted for speed or debugging reasons and has to be reverse engineered for major enhancements. Modifications in the source code should effect in the (re)constructed diagrams. Especially the paper presents a technique using annotations of the syntax graph to gain semantics knowledge. For that reasons, first static information in form of UML class-diagrams is reengineered and afterwards behavioural diagrams namely story-diagrams are recovered. We introduce a solution to inspect Java source code by defining fuzzy patterns. Fuzzy patterns are defined using collaboration diagrams. The huge number of implementation variants for code clichés results in a fuzzy pattern definition for each variant. To catch many variants in one fuzzy pattern definition we introduce fuzzyness into the definition. This fuzzyness let us deal with uncertainty coming from the generalisation of several similar code clichés. As detection mechanism for fuzzy patterns, we introduced Generic Fuzzy Reasoning Nets and presented a mapping of fuzzy pattern to GFRN's.

We are currently working on a more handy syntax and a first implementation of our approach. Therefore, we enhance the Fujaba system, which already supports the instantiation of design patterns and a rudimentary mechanism to extract design patterns out of Java source code. We plan to enhance our approach by inheritance and polymorphism like in object-oriented languages. For example, a fuzzy pattern detecting aggregations between classes could inherit from a general association detecting fuzzy pattern. Because typically, the difference between them is an explicit deletion of the aggregated objects when using aggregations.

The current prototype of the Fujaba environment is available as free software and comprises about 330 000 lines of pure Java code. Additional information and the current release version of the Fujaba system can be downloaded via:

`http://www.fujaba.de`

REFERENCES

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, 1999.

[DLP94] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 439–503. Clarendon Press, Oxford, 1994.

[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 Int. Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[HN90] M. T. Hanrandi and J. Q. Ning. Knowledge Based Program Analysis. In *Journal IEEE Software, volume 7, number 1*, pages 74–81, January 1990.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

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

[JH98] J.H. Jahnke and M. Heitbreder. Design Recovery of Legacy Database Applications based on Possibilistic Reasoning. In *Proceedings of 7th IEEE Intl. Conf. of Fuzzy Systems (FUZZ'98). Anchorage, USA.*. IEEE Computer Society Press, May 1998.

[JSZ97] J.H. Jahnke, W. Schäfer, and A. Zündorf. Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer Verlag, September 1997.

[JZ98] J.H. Jahnke and A. Zündorf. Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modeling. In *Proc. of 9th International Workshop on Software Specification and Design, Ise-Shima, Japan*, pages 77–86. IEEE Computer Society Press, 1998.

[KNNZ00]H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the 22th Int. Conf. on Software Engineering (ICSE), Limerick, Irland*. ACM Press, 2000.

[KSRP99]R.K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[NNSZ99]U. Nickel, J. Niere, W. Schäfer, and A. Zündorf. Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems. In *Proc. of Object-Oriented Modeling of Embedded Realtime Systems workshop (OMER)*. Technical Report 1999-01 University of the German Armed Forces Munich, 1999.

[NNWZ00]U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Technical Report University of Karlsruhe, 2000.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[Qui94] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[Rad99] A. Radermacher. Support for Design Patterns through Graph Transformation Tools. In *Proc. of Int. Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 1999.

[Rha] ILogix. *Rhapsody, the Rhapsody case tool. Online at http://www.ilogix.com*.

[Ros] Rational. *Rose, the Rational Rose case tool. Online at http://www.rational.com*.

[Tog] Object International. *TogetherJ, the TogetherJ case tool. Online at http://www.togethersoft.com.*

[Vin97] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[Wil94] L.M. Wills. Using Attributed Flow Graph Parsing to Recognize Programs. In *Int. Workshop on Graph Grammars and Their Application to Computer Science*, Williamsburg, Virginia, November 1994.