Fuzzy Logic based Interactive Recovery of Software Design*

(Extended Abstract)

Jörg Niere
Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100, D-33098 Paderborn
Germany
nierej@uni-paderborn.de

ABSTRACT

This abstract presents an approach to semi-automatically detect pattern instances and their implementations in a software system. Design patterns are currently best practice in software development and provide solutions for nearly all granularity of software design and makes them suitable for representing design knowledge. The proposed approach overcomes a number of scalability problems as they exist in other approaches by using fuzzy logic, user interaction and a learning component.

1. MOTIVATION

"*Never touch a running system*" is one of the most famous idioms in computer science and best practice for many software systems. This results from the experience that changing those software systems often has side-effects usually in parts that have not been adjusted to the change. The side-effects usually result from an awful documentation of the software system dependencies, because time-to-market has often a higher priority compared to an expensive good documentation of the system. Large and older systems, so-called *legacy systems*, often contain even no or only fragments of documentation. A re-documentation of those systems is usually very expensive, because it is mostly done manually.

Today, the *Unified Modelling Language* (UML) has become a standard for describing software systems. The UML consists of several different diagram types used for different purposes in a software development process. It is therefore naturally to use the UML diagrams also for re-documenting existing system, because they are common knowledge for nearly all developers and enable a seamless integration later in a redesign process.

In addition to the different diagram types in UML, *design patterns* [GHJV95] are best practice in software development. Design patterns introduced by Gamma et al., former known as Gang-of-Four (GoF) patterns, provide solutions for recurring problems. Today patterns of all granularity of software design exist in literature, e.g. implementation patterns, distribution patterns, architecture patterns and design patterns. If a pattern is used in an actual software system's design, it is called a *pattern instance.* Typically, there exist many different pattern instances for one pattern and even the actual implementation of a pattern instance can differ from one instance to another.

GoF-patterns provided by Gamma et al. are the result of an intensive (more or less manual) reengineering process of existing software systems at Big Blue. Consequently, GoF-patterns can be seen as a comprised collection of recurring successfully employed implementations made by independent developers in different software systems. This makes them highly suitable as a mean for legacy system understanding and as a representation of design knowledge. In addition, patterns connect several parts of a system which makes them ideal to document dependencies.

A GoF-pattern provides a solution for a problem in terms of a definition of the static and dynamic behaviour including usually one example and one implementation possibility. Thereby most parts of a GoF-pattern's description are informal which offers many interpretation opportunities. Typically, the static structure of a GoF-pattern is given as class diagram, while the behaviour is mostly described in prose and thereby not formally defined. To support an automated recognition of GoF-pattern and other pattern instances in existing software systems a formal definition of a pattern is indispensable.

In addition to a formal definition of a pattern the success of an automated recognition process of pattern instances highly depends on its scalability. Tools supporting an automated recognition of pattern instances must be able to analyse thousands or millions lines of code (LOC). The scalability is often strongly depending on the number of pattern definitions. Raising the scalability often means a reduction of the number of pattern definitions or relaxing the definitions in such a way that one pattern definition covers more than one pattern instance or implementation. Both, reducing the number of pattern definitions and relaxing the definitions, reduces the preciseness of the analysis where in the first case not all pattern instances are found and in the second case *false-positives* occur (erroneously recognized instances and implementations).

In practice it is impossible to run a fully automated analysis with a catalogue consisting of all pattern instance and implementation definitions for all patterns. Fortunately, for the analysis of a software system, it is usually sufficient to take only those patterns into account, which are relevant for the software system's domain. Focusing on a specific domain reduces the number patterns dramatically but does not solve the problem of a complete enumeration of all pattern instances and implementations of one pattern in one domain. Thus, a reengineering process has to be interactive where the engineer must be able to adapt a pattern instance or implementation definition to the actual system in a certain domain during the analysis. In addition, such an interaction

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

allows the engineer to infer personal hypothesises and presumptions and to integrate results from other analyses, e.g. original documentation or interviews with the developers of the system.

2. RELATED WORK

Other approaches related to the recovery of design patterns from a software system's implementation have failed for several reasons.

Approaches using only parts of an implementation, such as header files in C/C++ produce many false-positives, because several design patterns are structural identical but behavioural different.

Many approaches use deductive (bottom-up) execution mechanisms. This forces scalability problems, because they provide interesting results only after a complete analysis, which can be a long running process. In the worst case the analysis provides no or wrong results and it has to be performed once again.

Promising approaches use pattern matching approaches based on graph theory. Unfortunately, the sub-graph isomorphism problem, which occurs in those approaches, is NP-complete and more sophisticated algorithms have to be implemented manually, which is error-prone and hard to maintain.

Other approaches try to identify patterns by analysing code for recurring constructs. Those approaches are better suited for the detection of new design patterns, because they are not able to handle implementation results appropriately.

3. MY APPROACH

My approach presents a reengineering system including a process and techniques to extract pattern instances from a software system's implementation semi-automatically. Although the focus lies on GoF-patterns the approach can also be used to detect other kinds of pattern instances such as architectural, implementation or distribution patterns.

Graph grammars [Roz97] build the formal basis of the approach. Patterns are encoded as graph transformation rules and the to be analysed source code is parsed in its abstract syntax graph representation. The graph transformation rules are notated as UML collaboration diagrams, which are common knowledge and reduce the learn effort for other engineers.

Common parts in different patterns can be defined in separate rules and can be (re)used in the rules for the original patterns. Such common parts are called *sub-patterns* or *sub-rules*. In addition to the composition of patterns the approach supports also (structural) inheritance in object-oriented terms. Both raises the reuse and reduces the number of rule definitions. To handle the large number of implementation variants of a pattern instance, rules defining patterns and sub-patterns are enhanced with fuzzy values to describe a degree of uncertainty, cf. [ZK92]. With this uncertainty one rule can match for several implementations with a certain degree. This reduces the number of required rule definitions dramatically.

The detection algorithm uses a certain graph parsing technique, cf. [RS95], which annotates the abstract syntax graph of the source code for any found pattern. The algorithm uses a *forward/ backward chaining* (combined bottom-up, top-down) strategy. Such a combination accelerates providing first interesting analysis results. This is the major advantage in comparison to pure deductive approaches, which are usually able to provide interesting results only after a complete analysis.

Typically the found analysis results are uncertain corresponding to the fuzzy values defined in the rules. Uncertain results are accepted or rejected automatically when the uncertainty is higher or lower a certain bound or the engineer can accept and reject results manually. A learning component logs the interactions and recalculates the fuzzy values of the rules, and the acceptance and rejection bounds based on statistic analyses. This automates further analysis. In addition to the acceptance or rejection of uncertain results, the engineer is also able to adapt the rules to a certain domain during the analysis process. This becomes very efficient in combination with the early delivery of interesting results by the detection algorithm, because actions taken by the engineer always influence further analysis. For example, in case of an emergency, i.e. all instances are false-positives or no instance is found, the engineer can stop the analysis, investigate the current results and adapt the rules adequately in a very early analysis state.

The reengineering system is specified using UML and the Fujaba environment, cf. [FNTZ98]. Fujaba provides editors for UML class and activity diagrams and a code generation algorithm and is used to specify the reengineering system as well as its results. The effectiveness of my approach and the tools is shown analysing large systems. For example Java's Abstract Window Toolkit (AWT), the SWING library with about 200k LOC and Fujaba itself with more than one million LOC. All examples have been built using design patterns, though the evaluation is used to show the preciseness of my approach, extracting GoF-pattern instances and comparing them with the documentation of the systems. Using the reengineering system analysing a legacy system shows the application of my approach on foreign implementations. Thereby the advantages of integrating the engineer in the process are also investigated and discussed.

The current Fujaba analysis prototype is downloadable via http://www.upb.de/cs/fujaba and for more details on the algorithm and the evaluation see [NSW+02].

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.

[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. (to appear).

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[RS95] J. Rekers and A. Schürr. *A Graph Grammar Approach to Graphical Parsing*. In Proc. of the IEEE Symposium on Visual Languages, Darmstadt, Germany. IEEE Computer Society Press, 1995.

[ZK92] L.A. Zadeh and J. Kacprzyk. *Fuzzy Logic for the Management of Uncertainty*. John Wiley and Sons, Inc., 1992.