1. Abstract
Before redesigning a software system it is often necessary to extract design information from 
the system’s implementation, because the documentation is not up-to-date and highly differs 
from the system’s implementation. Manual analysis, which is current practice, is expensive and 
available tools can often not be applied to large systems. To overcome these problems, this re-
search abstract presents an approach to detect pattern instances and their implementations in a 
software system. Design patterns as e.g. the Gang-of-Four-patterns are currently best practice 
in software development and provide problem solutions for nearly all software design granu-
ularity, e.g. implementation patterns or architectural patterns. A pattern description is mostly in-
formal. To support an automated recovery of pattern instances my approach uses graph 
transformation rules enriched with fuzzy values to define patterns formally. I also developed 
an incremental and interactive algorithm which is flexible enough to analyse systems from dif-
ferent domains. An integrated learning component supports the reengineer by adapting the 
fuzzy values automatically based on the history of interactions. The approach overcomes a 
number of scalability problems as they exist in other approaches.

2. Motivation
“Never touch a running system” is, for sure, 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 raises problems a developer has not been expected, usually in parts 
that have not been adjusted to the change. There are various origins for this phenomenon but it 
usually results from an awfully maintained documentation of the software system. During the 
development and maintenance of a system its design documents differ more and more from its 
implementation because time-to-market has 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 they are 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 in a redesign process afterwards.

In addition to the different diagram types provided by the 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. Since the
GoF-patterns have been published, others have created new patterns and have discussed and published them on conferences and workshops. Thus, today exist patterns of all granularity for software design, e.g. implementation patterns, distribution patterns, architecture patterns and of course 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. Most interesting is the fact that especially the 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. Thus, GoF-patterns can be seen as a comprised collection of recurring 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.

**Precise Pattern Definition**

A GoF-pattern certainly 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 and open many gaps and interpretation opportunities. Typically the static structure of a GoF-pattern is given as an OMT [RBP+91] diagram, which is comparable with an UML class diagram, but the behaviour is mostly described in prose and thereby not defined formally. To support an automated recognition of GoF-pattern and other pattern instances in existing software systems a formal definition of a pattern is indispensable.

**Applicability to Large Software Systems**

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, i.e., doubling the number of pattern definitions often results in a more than doubled analysis run-time. 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).

**Adapting Patterns to a Specific Domain**

Another point, which is currently underestimated, is, that it is practically 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 typically 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 that it is also even impossible to enumerate all pattern instances and implementations of one pattern in one domain. Thus, a reengineering process has to be interactive where the reengineer 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 allows the reengineer also to infer personal hypotheses and presumptions and to integrate results from other analyses, e.g. original documentation or interviews with the developers of the system.
3. Related Work

There exist several approaches in the field of program analysis and program understanding in the literature, but, due to the lack of space, this section focuses on approaches which also try to detect pattern instances and which are related to the used techniques in my approach.

Comparable work on reengineering source code stems from Harandi and Ning [HN90] who present program analysis based on an Event Base and a Plan Base. A parser constructs rudimentary events from the source code to be analysed. Plans define the relationship between (incoming) event(s) and they fire a new event corresponding to the plans intention. Each plan corresponds to exactly one pattern variant instance or implementation, which lets the approach fail for large systems, because of the large number of different pattern instances or implementations for even one pattern. The same holds for the approach of Paul and Prakash [PP94]. Both approaches use a deductive execution semantics where in each deduction step pattern matching techniques are used. Thus, both approaches can be seen as a basis for other approaches.

Wills [Wil96] presents an approach to identify common computational structure such as searching or sorting algorithms. Wills uses also pattern matching techniques where patterns are encoded as rules stemming from a special graph grammar. The graph grammar combines control flow and data flow and is thereby comparable to Harandi and Ning’s plans. Unfortunately, an evaluation shows that her approach is only able to analyse a few thousand lines of code, because her pattern matching algorithm is NP-complete.

Jahnke [Jah99] presents a successful result analysing large relational database systems. He integrates the reengineer in his analysis process and is able to handle the large search space. In addition, possibilistic logic integrated in his Generic Fuzzy Reasoning Nets (GFRNs) handles uncertainty. GFRNs use so-called clichés as irrevocable facts, which are detected in the database’s code using a classical pattern matching approach. This is sufficient for the analysis of relational database systems but fails for the recovery of pattern instances or implementations. Applying the process as well as the GFRNs to the recovery of some easy implementation pattern instances fail, because there exist only a few number of cliché instances of one cliché in comparison to the large number of pattern instances and implementations, cf. [JNW00].

Concerning the detection of design patterns, Kraemer and Prechelt [KP96] present an approach of analysing C/C++ source code to extract design patterns. Hence, they analyse the header files (structural parts) only, they get many false-positive, because many design patterns are structural identical but behavioural different.

Analysing behaviour as well as structure using patterns is presented by Keller et al. in [KSRP99]. A common abstract syntax graph model for UML is used to represent source code as well as patterns. Matching the syntax graph of a pattern on the syntax graph of the source code is done using scripts. The scripts have to be implemented manually. The definitions in such a script language become difficult to maintain and to reuse. It is also not possible to modify a script, which means an adaptation of a pattern to a certain domain, during the analysis run-time. Thus, the approach fails for the analysis of unknown software systems.

Tonella and Antoniol’s [TA99] approach of detecting patterns is orthogonal to the other ones. They do not use pre-defined patterns but try to find a pattern instance in source code but analyse the code for recurring constructs. Statistic evaluation summarizes the analysis result and present which construct occurs how many times in the program. On the one hand this approach seems to be more useful for the detection of (new) patterns than for pattern instances but could be used to identify unknown pattern instances or implementations in unknown systems. Categorizing the found results and comparing them to existing patterns is, unfortunately, hard work and has to be done manually.
4. My Approach

This research presents a reengineering system including a process and techniques to extract pattern instances from a software system’s implementation. Thereby the focus lies on GoF-patterns but the approach is not limited to them but can also be used to detect other kinds of pattern instances such as architectural, implementation or distribution patterns. The core is an interactive pattern matching algorithm based on graph grammars [Roz97], fuzzy logic [ZK92] and graph parsing techniques [RS95].

Patterns are encoded as graph transformation rules, which are a part of graph grammars and deal as formal definition language. 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* and reduce the complexity\(^1\). In addition to the formal basis, the rules are notated as UML collaboration diagrams which reduces the learn effort for other reengineers.

The used graph parsing technique allows the algorithm to provide a *forward/backward chaining* (combined bottom-up, top-down) analysis, which accelerates providing first interesting analysis results and reduces the overall analysis time. The former is the major advantage in comparison to pure deductive approaches, which are usually able to provide uninteresting intermediate results, quickly.

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. With this uncertainty one rule can match for several implementations with a certain degree. This reduces the number of rule definitions and raises the scalability. Uncertain results can be accepted or rejected by the reengineer. A so-called learning component logs the interactions and recalculates the fuzzy values of the rules at a certain time based on statistic analyses.

In addition to the acceptance or rejection of fuzzy valued results, the reengineer is also able to adapt the rules to a certain domain during the analysis run-time. This becomes very efficient in combination with the early delivery of interesting results by the algorithm, because actions by the reengineer always influence further analysis. For example, in case of an emergency, i.e. all instances are false-positives or no instance is found, the reengineer 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 [FNTZ98, KNNZ00], which is able to generate Java source code from a specification. Consequently, appropriate tool support is automatically given. 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 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 reengineer in the process are also investigated and discussed.

Note, in our accepted paper [NSW+02] for this ICSE we present an excerpt of this research. The paper introduces the formal definition of patterns using graph transformations, the algorithm with its forward and backward chaining, a rudimentary interaction with the reengineer and first evaluation results extracting design patterns from the AWT library. The paper does not include the fuzzy logic aspects with all its consequences, i.e. the influence on a pattern’s definition and the algorithm, additional interaction possibilities and the learning component.

---

1. The approach provides also (structural) inheritance, which raises the reuse and reduces the number of rule definitions, too.
References


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


