1. Introduction

Today, software engineers spend most of their time maintaining software systems. Appropriate (design) documentation is often missing or no longer consistent with the actual implementation. Especially in case of legacy systems which have not been developed using a model-driven approach, it is the system’s source code that has to be understood and changed eventually, which is a time consuming and expensive task.

Design patterns are good design solutions to recurring problems. In contrast, anti patterns [2], bad smells [4, 5], or violations of design heuristics [9] describe recurring bad design solutions.1 Data classes, for example, consist only of fields and almost no behavior, which means the behavior belonging to the data may be spread throughout the system. Huge god classes assume too much responsibilities in a (sub-)system. Other examples are conditional dispatchers, i.e. huge conditionals for handling requests, or conditionals in combination with fields to represent different states/types of an object. Anti pattern instances may complicate maintenance activities significantly and thus knowledge about them is crucial for the planning of reengineering activities.

Anti patterns may usually be transformed into better solutions without changing the original behavior of the system by means of refactoring [4]. In case of a data class the behavior belonging to the data should be moved into the class. God classes should be split into several smaller classes [9]. Conditional dispatchers may be transformed into a solution based on the chain of responsibility or command design patterns [5].

When improving anti pattern instances, usually the goal is to improve the structure of the system, but to leave its behavior unchanged. In case of refactorings the behavior preservation is ensured by checking preconditions before their application. However, there is no guarantee as to whether the preconditions are sufficient or the refactoring itself does not break the behavior somehow. Thus, extensive testing is recommended when doing refactoring which of course can not guarantee behavior preservation either.

Anti patterns are described mainly informally in the literature as are the transformations for their improvement. Consequently, detection and improvement are done manually to a large extent. The goal of this research is to provide a tool-supported reengineering approach which allows for the formal specification and automatic detection of anti pattern instances in the source code of a software system as well as the formal specification of executable transformations for the improvement of anti patterns and their (manually triggered) application to detected anti pattern instances. Furthermore, the approach will support the semi-automatic verification of the transformations to fulfill specified criteria. Those may be criteria which are necessary for behavior preservation (but not sufficient, however).

Some approaches, e.g. [7], try to recognize anti patterns based on metrics. Metrics alone, however, are not able to cover semantic aspects and thus fail to recognize more complex anti patterns. None of the approaches addresses the improvement of their findings. In [6] the formalization of refactorings based on graph transformation is proposed. In addition, they propose the specification of well-formedness rules and behavior preservation criteria which are necessary to hold (but not sufficient) for behavior preservation. However, the formalization of such criteria alone is not sufficient. Especially when refactorings become more complex, a verification that they do not violate the criteria is needed.

2. Approach

The approach can be divided into three major steps: anti pattern recognition, transformation, and transformation verification.
**Anti Pattern Recognition:** The recognition step is based on an approach for the recognition of design pattern instances which has been developed in our group [8]. The approach is based on an extended Abstract Syntax Graph (ASG) representation of a system’s source code. Patterns, in this case anti patterns, are formally specified by graph grammar rules with respect to the ASG, which we call pattern rules. A pattern rule defines an ASG node structure which has to exist in the ASG representation and adds an annotation node to indicate a found pattern. The pattern rules may contain annotation nodes created by the application of other pattern rules which enables the composition of pattern rules to describe more complex structures. In addition, pattern rules may require ASG nodes representing e.g. classes or methods to have certain metric values.

The source code is parsed into the ASG representation and the pattern rules are applied to the ASG by an inference algorithm [8]. Successful rule applications produce pattern instance candidates which are marked in the ASG by annotations.

**Anti Pattern Transformation:** The improvement of anti patterns is formally specified by so-called transformation rules. The transformation rules are specified as graph grammar rules w.r.t. the ASG (including annotations created by the recognition) based on Story Diagrams [3]. Story Diagrams are basically extended UML Activity Diagrams in which the activities are specified as graph grammar rules. The transformation rules will permit sequences of activities and loops and they may call each other to facilitate composition and reuse.

In the anti pattern transformation step the reengineer manually reviews the candidates identified by the recognition step and decides which transformations are to be applied to which candidate, if any. The transformations are then executed automatically which results in the transformed source code.

**Transformation Verification:** Before transformation rules are applied, they may be verified. For the verification, forbidden or to be preserved source code patterns will be specified as graphs w.r.t. the ASG. The patterns may be used to specify criteria which are necessary for behavior preservation. An example for such a (forbidden) pattern is an identifier inside the body of one method which is bound to a parameter of a different method (which may happen when a statement is extracted into a new method). The goal is to verify that the transformation rules do not create (or destroy) such forbidden (or to be preserved) patterns.

Recently, an approach for the automatic verification of graph grammar rules to not create forbidden graph patterns has been developed in a different context in our group [1]. The approach is currently capable of verifying single graph grammar rules with negative application conditions. The transformation rules, however, contain several graph grammar rules embedded in control flow. A separate verification of each of the embedded rules would require forbidden patterns to not exist after each of them, which is too restrictive. Instead forbidden patterns must not exist before and after the application of a complete transformation rule, but are allowed in between. The idea is therefore, to adapt the approach to the verification of the more complex transformation rules, which is mainly current work.

3. **Conclusion**

The described approach supports the detection of anti pattern implementations in source code. Thus, it can be used for the evaluation of existing software in the planning stage of reengineering activities. In addition, the approach supports the actual reengineering by facilitating the improvement of anti pattern instances by transformations and the verification of those transformations.

**References**


