CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs

Dirk Beyer and Claus Lewerentz
Software Systems Engineering Research Group
Technical University Cottbus, Germany
{db | cl}@informatik.tu-cottbus.de

1. Introduction

The engineer in a design analysis process has two major objectives: he has to comprehend the architecture and the design of the system, and he has to assess the quality of the software system. Both tasks need effective tool support for todays large software systems.

In the **comprehension** process, the engineer has to identify structures which are important for the understanding of the design. These structures can be described by patterns. The most famous example for such patterns are the object-oriented design patterns [6], which represent good design solutions on a more abstract level, or anti-patterns, describing problematic program structures (cf. bad smells [5]). The detection of such structures considerably supports design comprehension.

In the context of this paper, **patterns** are formally defined using the notion of partial subgraphs and their relational specification.

Patterns can be helpful also for **quality assessment** of the design. By defining anti-patterns which represent problematic pieces of design and by identifying the instances of such patterns automatically, the process of assessment can be accelerated. Patterns for design weakness which should be inspected are e.g. cycles in the call graph, role identity of classes, degenerate inheritance, and "curious" superclasses. From recognized design weaknesses the engineer can derive hints for the improvement of the quality in a restructuring phase.

Automatic pattern-based recognition of design weakness is a research topic since almost 10 years. Reports about experiments with existing approaches reveal two major problems: A notation for **easy and flexible specification** of the pattern is missing; only a restricted set of patterns is applicable because of the limitations of the specification language. **Performance improvement** is needed, because the computation time of existing tools is to high to be acceptable for large real-world systems.

2. The tool CrocoPat

The tool *CrocoPat* satisfies the following three requirements: **(1)** The analysis is done automatically by the tool, i.e. without user interaction. **(2)** The properties of a system are specified in an easy and flexible way because the patterns are described by relational expressions. On demand the user is able to define new patterns he is interested in, or to change existing patterns to solve specific problems. **(3)** The tool is able to analyze large object-oriented programs (1'000 to 10'000 classes) in acceptable time.

In terms of graph theory, the tool *CrocoPat* does subgraph search. In terms of relational algebra, the tool searches for tuples fulfilling a given predicative expression. The approach is not bound to a specific meta model of the program: the expressions are based on standard operators and the tool does not use the meaning of the relations for analysis. However, the call relation and the inheritance relation on the three levels of packages, classes, and methods are often sufficient for the design recovery (cf. [4]).

All relations are represented by **binary decision diagrams (BDDs)** [3]. BDDs give canonical and compact representations of sets and allow for an efficient implementation of operations like intersection, union and existential quantification. More details about the tool and the pattern specification language are documented in [1].

3. Example applications

GoF design patterns [6]. The use of design patterns indicate good design because these pattern are known to support flexible and understandable structures. Thus, to support design understanding it can be helpful to find instances of design patterns within an object-oriented program. For identifying all instances of the Composite pattern the computation of all tuples (x, y, z, l) is necessary, with x is a Client class, y is the Component class, z is a Composite class, and l is a Leaf class of the pattern, i.e. $(x, y) \in Call \wedge (z, y) \in Inherit \wedge (z, y) \in Contain \wedge ((l, y) \in Inherit \wedge (l, y) \notin Contain)$.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

Circle. A class x should be understandable independently from the classes which call a method of class x. To understand a class, we have to understand all classes which it uses directly or indirectly. If one of those classes is the class itself then the understanding is complicated. Circles can be introduced during the evolution of a program if further functionality is added. The experience shows that the number of circles decreases during restructuring activities. Thus, for analyzing the occurrence of circles in the call relation we have to compute all tuples (x) with class x occurs in a circle.

Role identity. Another interesting design analysis question is whether there exist classes with identical roles, i.e. two classes use the same classes and are used by the same classes. Classes with identical roles occur in polymorphic design structures as subclasses or when an old class is replaced by a new one, but the old class is not removed from the object-oriented program.

Degenerate inheritance. The wrong use of inheritance often leads to misunderstandings and bad design. Let X be a class which implements an interface Y, and class S 'extends' class X *and* 'implements' interface Y. We have to pay attention to such design structures because S does not really implement the interface Y, rather it uses the implementation given by class X. One suggestion would be to omit the 'implements' relation to interface Y. The design question is whether the direct inheritance is redundant or not. We search for a set of classes $\{y, x, u\}$ with the condition $(x, y) \in Inherit^+ \wedge (u, x) \in Inherit \wedge (u, y) \in Inherit$, i.e. a subclass inherits directly and indirectly from a superclass.

Curious superclasses. The goal of separating the interface from its implementation is that the interface (superclass) should not know anything about their implementation (subclass). Thus, we have to pay attention to superclasses which call or contain instances of their subclasses.

4. Summary

CrocoPat is a new tool for efficient pattern-based analysis of large object-oriented programs. Patterns can be **flexibly specified** by expressions based on standard mathematics provided by the tool language. It is easy to specify patterns in different variants in a compact form, adapted to specific situations.

The software system which is to be analyzed is interpreted in terms of relations, and the patterns are described by relational expressions over these relations. The tool represents the abstract model of the program using a data structure based on binary decision diagrams, which are proved to allow for an efficient recognition also for large systems comprising several MLOC source code.

System	No. of classes	LOC	Closure
Mozilla	4'818	3'236'875	73 s
JWAM	999	167'178	3.0 s
wxWindows	378	217'832	1.1 s

Table 1. Performance of transitive closure computation for some example systems

The tool can help to improve the productivity of comprehension and assessment processes by using it in combination with other tools for program analysis. We use the tool in combination with tools for software measurement and navigation [2] and software visualization [7].

To demonstrate the performance of the tool, Table 1 reports the computation times of constructing the transitive closure of the call relation for some example systems. Each row of the table indicates the name of the system, the number of classes of the system, the number of lines of code (LOC) and the time needed to compute the transitive closure of the call relation (TC). The computation times are obtained on a Pentium III processor with 850 MHz and 50 MB RAM for the BDD package. The most complex operations are the computation of transitive closures and complements of large relations. Both operations are not supported by SQL-based approaches (cf. *QualiT* [2]) and can not be computed efficiently using the Prolog-based approaches (cf. *Goose* [4]).

References

[1] D. Beyer and C. Lewerentz. CrocoPat: A tool for efficient pattern recognition in object-oriented programs. Technical Report I-04/2001, BTU Cottbus, 2003.

[2] W. R. Bischofberger. *QualiT: User's Guide and Reference Manual*. Software Tomography GmbH, http://www.software-tomography.com, 2003.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1986.

[4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pages 18–32. IEEE Computer Society, 1999.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, 1999.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP 1993)*, LNCS 707, pages 406–431. Springer-Verlag, Berlin, 1993.

[7] C. Lewerentz and A. Noack. CrocoCosmos – 3D-visualization of large object-oriented programs. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*. Springer-Verlag, 2003.