CrocoPat: A Tool for Efficient Pattern Recognition in Large Object-Oriented Programs *

Dirk Beyer and Claus Lewerentz

Software Systems Engineering Research Group
Technical University Cottbus
D-03013 Cottbus, Postfach 10 13 44, Germany
Computer Science Reports I-04/2003, January 2003
{db | cl}@informatik.tu-cottbus.de

Abstract. Nowadays, software systems are too large to be understandable by reading the source code. For reengineering activities, methods and tools for automated design recovery are needed. *CrocoPat* is a tool for efficient pattern-based design analysis of object-oriented programs. Patterns can be flexibly specified by expressions based on standard mathematics. The software meta model 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 for performance improvement. The representation is proved to allow for an efficient recognition also for large systems up to 10'000 classes comprising several MLOC source code.

Keywords. Software comprehension, Quality assessment, Pattern recognition, BDD

1 Motivation

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 [GHJV93], which represent good design solutions on a more abstract level, or anti-patterns, describing problematic program structures (cf. bad smells [Fow00]). The detection of such structures considerably supports design comprehension.

In the context of this paper, the notion **pattern** is used for a specification of a piece of design for which the engineer wants to know whether instances exist in the program.

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.

* This paper is the extented version of [BL03], which is published in the proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003, Portland).

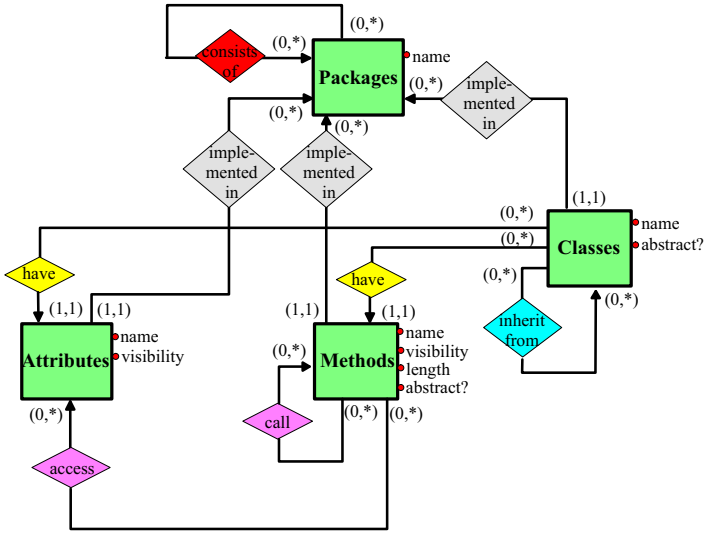

Fig. 1. Meta model for object-oriented programs

The rest of the paper is organized as follows: At first we give a short overview of our tool; in Sect. 3 we define the new pattern specification language. To evaluate our approach, Sect. 4 demonstrates the method and the performance of the tool implementation on some practical examples. In Sect. 5 we discuss some existing approaches to the problem, and at the end of the paper we summarize and discuss our approach.

2 Tool Overview

To provide a solution for the problems mentioned in the introduction, the tool *CrocoPat* satisfies the following three requirements:

- The analysis is done automatically by the tool, i.e. without user interaction.
- 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.
- 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 so 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. [Ciu99]).

For the structural analysis we have to use an abstract representation of the program. This product abstraction is defined by a meta model. Figure 1 shows the meta model which we use in our assessment projects. Using this abstraction we can regard the program as a set of relations. Some of them are a directly included in the abstract representation (e.g. method call, attribute access), other useful relations we can derive from these relation (e.g. call relation and containment relation on class level).

All relations are represented by **binary decision diagrams (BDDs)** [Bry86] to achieve a crucial improvement of the analysis performance. BDDs give canonical and compact representations of sets and allow for an efficient implementation of operations like intersection, union and existential quantification.

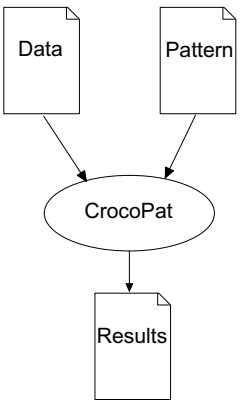

Fig. 2. The tool CrocoPat

The tool *CrocoPat* works on two inputs: the abstract product data and the description of analysis tasks (i.e. mainly the pattern definition) as shown in Fig. 2. After starting the tool no user intervention is required until the tool returns with notifying about the instances of the pattern it has found. The procedure of analysis using *CrocoPat* consists of the following steps:

1. *Extract data from the source code.* A program analysis tool like *Sotograph* [Bis03] is used to extract automatically all relevant data regarding to the meta model from the source code and to store the data in a relation file.
2. *Create pattern definition.* The pattern of interest has to be defined using the pattern specification language described in Sect. 3 on the basis of the relations which are stored in the relation file in step 1.
3. *Run analysis. CrocoPat* starts with reading the relations from the relation file and transforming them to the corresponding BDD representation. Then it applies all expressions specified in the pattern definition to obtain the pattern instances as result.

The tool introduced in this paper is interesting especially for the analysis of programs regarding to design guidelines which are not easy to check using a software measurement tool. Guidelines like 'a class should not contain more than 7 attributes' can be checked comfortably also using existing measurement tools like *Datrix* [ML96], *Crocodile* [SL97], or *Sotograph* [Bis03]. The tool *CrocoPat* is designed especially to recognize patterns for which more complex computations are required to find all its instances.

3 Pattern Specification with *CrocoPat*

The language for the specification of patterns is one of the most important parts of a pattern-based approach to design analysis. The description has to be *independent from technical details* of the implementation, i.e. the user does not have to know how the tool works internally. The specification should be *flexible* enough to enable changes of the patterns during the assessment phase and the

repeated definition of new patterns. The notation has to be *easy to understand* to be used not only by few experts and has to be *powerful* enough to express all patterns the analyst wants the define.

The expressions of the relational algebra are fulfilling these requirements: they are abstract enough to have no implementation details in it, and they are easy to understand because only elementary mathematical expressions are allowed. In the following we define the syntax and semantics of the expressions.

Let the universe $\mathcal{U}$ be the set of all values[1] and X be a finite set of **attributes** (variables). A **tuple** t **of** X **(attribute assignment)** is a total function $t : X \to \mathcal{U}$. $Val(X)$ is the set of all tuples of X. Using a fixed ordering of the attributes in X we can use the vector notation $(t(x_1), \ldots, t(x_n))$ for a tuple $t \in Val(X)$. $R \subseteq \mathcal{U} \times \cdots \times \mathcal{U}$ is a **relation**. $\mathcal{R}$ is a finite set of relations. An **expression over** $\mathcal{R}$ **and** X is an element from the set $\Phi(\mathcal{R}, X)$, the set of all expressions over $\mathcal{R}$ and X. The set $\Phi(\mathcal{R}, X)$ is generated by the following grammar:

$$\varphi := R(x_1, \ldots, x_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid (\varphi) \mid x < x' \\ \mid x = x' \mid x = c \mid \exists(x, \varphi) \mid \varphi[x/x'] \mid TC(\varphi, x, x')$$

with $R \in \mathcal{R},\ x, x', x_1, \ldots, x_n \in X$ and $c \in \mathcal{U}$.

The **semantics of an expression** is given by the interpretation function $[\![\,.\,]\!] : \Phi(\mathcal{R}, X) \to 2^{Val(X)}$, which is defined inductively as follows:

$$\begin{aligned}
[\![R(x_1, \ldots, x_n)]\!] &:= \{t \in Val(X) \mid (t(x_1), \ldots, t(x_n)) \in R\} \\
[\![\neg\varphi]\!] &:= \{t \in Val(X) \mid t \notin [\![\varphi]\!]\} \\
[\![\varphi \wedge \varphi']\!] &:= [\![\varphi]\!] \cap [\![\varphi']\!] \\
[\![\varphi \vee \varphi']\!] &:= [\![\varphi]\!] \cup [\![\varphi']\!] \\
[\![(\varphi)]\!] &:= [\![\varphi]\!] \\
[\![x < x']\!] &:= \{t \in Val(X) \mid t(x) < t(x')\} \\
[\![x = x']\!] &:= \{t \in Val(X) \mid t(x) = t(x')\} \\
[\![x = c]\!] &:= \{t \in Val(X) \mid t(x) = c\} \\
[\![\exists(x, \varphi)]\!] &:= \{t \in Val(X) \mid \exists t' \in [\![\varphi]\!] : \forall x' \in X \setminus \{x\} : t(x') = t'(x')\} \\
[\![\varphi[x/x']]\!] &:= [\![\exists(x, \varphi \wedge x = x')]\!] \\
[\![TC(\varphi, x, x')]\!] &:= \{t \in Val(X) \mid \exists t' \in [\![\varphi]\!] : \exists t'' \in [\![TC(\varphi, x, x')]\!] : \\
&\qquad t(x) = t'(x) \ \wedge\ t'(x') = t''(x) \ \wedge\ t''(x') = t(x')\}
\end{aligned}$$

with $R \in \mathcal{R}, x, x', x'', x_1, \ldots, x_n \in X$ and $c \in \mathcal{U}$. φ is interpreted as the set of all tuples over X which fulfill the predicate φ.

Note. We use the notion 'tuple' corresponding to the theory of relational databases. The concept of attribute assignments is chosen because the identification of an element of a tuple by the attribute name (like column names in relational databases) is more convenient than by its position in the tuple.

All relations contained in or derived from the user-specific meta model can be used as base relation in *CrocoPat*. The most important relations are the call relation, the inheritance relation and the containment relation on the class level of abstraction. These relations can be represented by the following tuple sets $Call, Inherit, Contain \subseteq Val(X)$ (on the class level of abstraction $\mathcal{U}$ is the set of classes, $x \in X$ is the referencing class, $y \in X$ is the referenced class):

$$\begin{aligned}
Call &= \{t \in Val(X) \mid \textit{A method of class } t(x) \textit{ calls a method of class } t(y).\} \\
Inherit &= \{t \in Val(X) \mid \textit{Class } t(x) \textit{ inherits from class } t(y).\} \\
Contain &= \{t \in Val(X) \mid \textit{Class } t(x) \textit{ contains an instance of class } t(y).\}
\end{aligned}$$

[1] For analysis on the class level of abstraction we use unique class identifiers as values.

4 Evaluation of the Approach

In the following we explain some example patterns and their definition. We apply them to three example programs: Mozilla, JWAM, and wxWindows. Mozilla is a large C++ system (i.e. the code base) for the development of the internet application Netscape Communicator (4'818 classes, 3'236'875 LOC), JWAM is a Java framework based on the tools and materials approach (999 classes, 167'178 LOC), and wxWindows is a GUI framework implemented in C++ (378 classes, 217'832 LOC). We report also the performance results by giving a table with computation times for recognizing all instances of the patterns in these object-oriented programs at the end of this section.

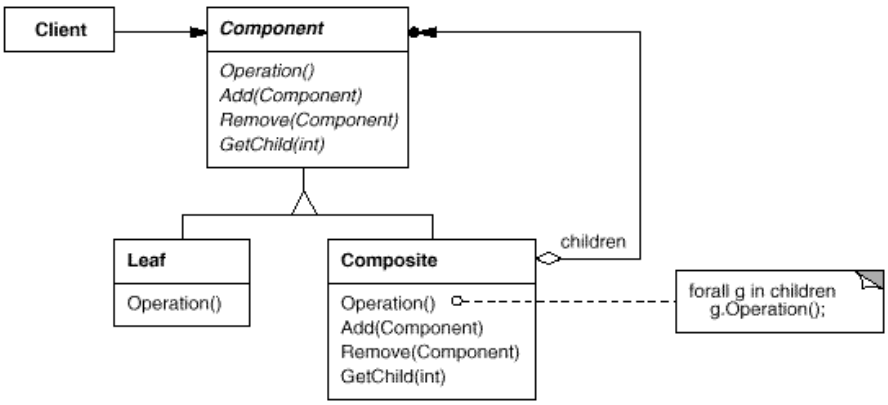

Fig. 3. Composite pattern [GHJV93]

GoF Design Patterns [GHJV93]. 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. To give an example, Figure 3 displays the UML diagram of the Composite design pattern [GHJV93]. For identifying all instances of this 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. $(x, y) \in Call \wedge (z, y) \in Inherit \wedge (z, y) \in Contain \wedge ((l, y) \in Inherit \wedge (l, y) \notin Contain)$. For the *CrocoPat* tool we have to use the expression $Call \wedge Inherit[x/z] \wedge Contain[x/z] \wedge (Inherit[x/l] \wedge \neg(Contain[x/l])$ to describe this pattern (cf. Sect. 3). Table 1 reports the number of instances of the design patterns Composite and Mediator in the example programs.

Pattern	Composite	Mediator
Mozilla	15	28
JWAM	14	6
wxWindows	4	5

Table 1. Number of composites and mediators in the analyzed systems

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 using the *CrocoPat* expression $\exists(y, TC(Call, x, y) \wedge (x = y))$.

Measure	Classes in circles
Mozilla	792
JWAM	30
wxWindows	63

Table 2. Number of classes which occur in some circle

Circles of length	2	3	4	5
Mozilla	338	40	5	2
JWAM	10	2	0	0
wxWindows	28	2	1	0

Table 3. Number of circles of varying length

With the help of some further specialized pattern expressions it is possible to compute the number of circles of some fixed length. Table 2 reports the number of classes which occur in some circle, and Table 3 lists the number of circles of length 2 to length 5 for each of the analyzed systems.

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.

For searching pairs of classes (x, z) which are identically embedded we can apply the following pattern, let $Call^{-1}$ be the tuple set $Call[x/tmp][y/x][tmp/y]$:
$\neg\big(\exists(y, Call \wedge \neg Call[x/z]) \vee \exists(y, \neg Call \wedge Call[x/z])\big) \wedge$
$\neg\big(\exists(y, Call^{-1} \wedge \neg Call^{-1}[x/z]) \vee \exists(y, \neg Call^{-1} \wedge Call^{-1}[x/z])\big)$

Table 4 shows the number of classes which are identically embedded within the three example systems.

Pattern	Rule identity
Mozilla	250
JWAM	9
wxWindows	12

Table 4. Number of identically embedded classes

Pattern	Rhombus	DegenInh	CuriousInt
Mozilla	918	39	25
JWAM	676	68	1
wxWindows	8	0	3

Table 5. Degenerate inheritance and curious superclasses

Degenerate Inheritance. The wrong use of inheritance often leads to misunderstandings and bad design [BLS01]. 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, instead it (potentially) 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.

In a measurement-based quality assessment of the Java Framework JWAM (cf. [BLL$^+$02] for an overview of the assessment project) one of the restructuring recommendation (No. 49) was to avoid such design structures. Using the measurement-based approach only one instance of this pattern was found. The pattern-based approach reveals that the number of such design structures is dramatically higher than assumed.

An interesting question for an object-oriented program is whether there exist such degenerative inheritance structures: we search for a set of classes $\{y, x, s\}$ with the condition $(x, y) \in Inherit^+ \wedge (s, x) \in Inherit \wedge (s, y) \in Inherit$, i.e. a subclass inherits directly and indirectly from a superclass. To compute all tuples (y, x, s) fulfilling this condition, we can compute the *CrocoPat* expression $TC(Inherit, x, y) \wedge Inherit[x/s][y/x] \wedge Inherit[x/s]$.

A derived pattern of this kind is the *rhombus-like inheritance* [BLS00,BLS01]. Designs with such inheritance structures are difficult to understand, because on the UML level it is not decidable which methods or attributes are visible in the subclass (even if the source code is present it is not obvious to most C++ programmers). In Java, rhombus-like inheritance means that one of the inheritance relations is a real 'extends' and the other is an 'implements' relation; so this pattern does not lead to problems in Java. Searching for rhombus-like inheritance means to identify sets of classes $\{y, x, v, u\}$ with the condition $(x, y) \in Inherit^+ \wedge (v, y) \in Inherit^+ \wedge (u, x) \in Inherit \wedge (u, v) \in Inherit \wedge x \neq v$. To compute all tuples (y, x, v, u) fulfilling this condition, we have to compute the *CrocoPat* expression $TC(Inherit, x, y) \wedge TC(Inherit[x/v], v, y) \wedge Inherit[x/u][y/x] \wedge Inherit[x/u][y/v] \wedge \neg(x = v)$. Table 5 reports the number of rhombi and the number of occurrences of degenerate inheritance (DegenInh) within the inheritance structure.

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 (direct or indirect) subclasses.

We can find such pairs of classes using the following pattern expression: $TC(Inherit[x/tmp][y/x][tmp/y], y, x) \wedge (Call \vee Contain)$ results in a set of tuples (x, y) with x is a superclass of y and x calls or contains an instance of y. The results are reported in Table 5 (CuriousInt).

Performance Results. Table 6 reports the performance results of computing the transitive closure – the most complex operation – of the call relation for some example system. 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 (Time Closure). The computation times are obtained on a Pentium III processor with 850 MHz and 50 MB RAM for the BDD package. Table 7 indicates the performance of recognizing all instances of the patterns Circle, Role identity, design pattern Composite, and Rhombus-like inheritance.

Measure	Classes	LOC	Tuples in call relation	Time Closure
Mozilla	4'818	3'236'875	13'423	73 s
JWAM	999	167'178	3'142	3.0 s
wxWindows	378	217'832	717	1.1 s

Table 6. Characterization of size and performance of transitive closure computation for three example systems

Pattern	Circle	Role identity	Composite	Rhombus
Mozilla	143	129	23	21
JWAM	3.3	5.0	3.1	2.1
wxWindows	1.3	2.1	1.0	1.1

Table 7. Computation times (in seconds) for some example patterns

5 Related Work

Automatic pattern-based recognition of design weakness is a research topic since almost 10 years. Some of the research results in this area are mentioned in the following.

SPOOL (Uni Montreal) is a tool for design navigation that supports browsing and searching the elements and relationships of an object-oriented progam [RSK00]. The tool is able to identify design pattern like the Factory Method.

Goose (FZI Karlsruhe) is a reengineering tool for investigation of the static structure of an object-oriented program [Ciu99]. A Prolog knowledge base is created from the source code and afterwards it can help to identify patterns specified in the Prolog query language. The tool has performance problems with complex operations like transitive closure computation and if a query is not optimized by a Prolog expert. **Pat** (Uni Karlsruhe) is a tool similar to Goose. It uses also the Prolog system for identifying pattern instances [KP96].

VizzAnalyser (Uni Karlsruhe) is an approach which uses not only static, but also dynamic analysis for searching behavioral patterns [HHL02]. The strategy is to use dynamic analysis for further restriction of the results of the static analysis to reduce the number of false-positives.

FUJABA (Uni Paderborn) integrates a semi-automatic approach to recognize instances of a specified pattern [NSW+02]. The recognition algorithm works incrementally and is based on the graph transformation system of the FUJABA project.

Sotograph (Software Tomography GmbH, Cottbus) is a program analysis workbench [Bis03]. The data of the product model are stored in a relational database. The main application of the tool is to analyze programs using software measurement techniques. It is also capable to find instances of patterns which can be defined using SQL statements. However, it is not possible to compute the complement or transitive closure of some relation, i.e. the class of definable patterns is clearly restricted.

RelView (Uni Kiel) is a general tool for computations on graphs [BBMS98]. It represents relations by BDDs, but is restricted to binary relations. Algorithms can by specified by operations of the relational algebra. This approach was inspiring the development of our own BDD-based tool which supports arbitrary relations and which is optimized for the application area of program design recovery.

6 Conclusion

CrocoPat is a new tool for efficient pattern-based analysis of large object-oriented programs. It 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 [Bis03] and software visualization [LN03]. Pattern-based recognition of design weakness and identification of design structures to support the understanding of large object-oriented programs are not new. The contribution of *CrocoPat* is to provide a very **flexible specification language** for pattern definition and to provide **high performance analysis** algorithms which are efficient even for large systems by using a BDD-based representation of relations.

Patterns can be flexibly specified by relational algebra expressions. 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 enable the efficient recognition also for large systems comprising several MLOC source code.

As the computation times in Table 7 show, the most complex operations are the computation of transitive closures (e.g. in pattern *Circle*) and complements (e.g. in pattern *Role identity*) of large relations. Both operations are not supported by SQL-based approaches (cf. *Sotograph* [Bis03]) and can not be computed efficiently using the Prolog-based approaches (cf. *Goose* [Ciu99]).

To improve the usability of our tool it would be nice to integrate it into other program analysis tools. It is planned to integrate *CrocoPat* into the *Sotograph* tool environment.

Further performance improvement is possible by implementing an operation cache to compute the same results of expensive operations (especially transitive closures) only once. Another interesting extension would be a client/server architecture to store BDDs for relations persistently in an independent server process. This would reduce the effort for loading the relations into BDDs for each computation separately, as shown by the Mozilla results: the computation time for computing *Composite* and *Rhombus* consists mainly of loading the relation (20.5 s).

Acknowledgements

We thank Andreas Noack and Ulf Milanese for applying the tool RelView to software graphs. The results of their experiments indicated the issues we had to address and to improve.

References

[BBMS98] Ralf Behnke, Rudolf Berghammer, Erich Meyer, and Peter Schneider. RELVIEW – a system for calculating with relations and relational programming. In Egidio Astesiano, editor, *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE 1998)*, LNCS 1382, pages 318–321. Springer-Verlag, 1998.

[Bis03] Walter R. Bischofberger. *Sotograph: User's Guide and Reference Manual*. Software Tomography GmbH, http://www.software-tomography.com, 2003.

[BL03] Dirk Beyer and Claus Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003, Portland)*. IEEE Computer Society Press, 2003.

[BLL+02] Holger Breitling, Claus Lewerentz, Carola Lilienthal, Martin Lippert, Frank Simon, and Frank Steinbrückner. External validation of a metrics-based quality assessment of the JWAM framework. In *Tagungsband des Workshops der GI-Fachgruppe 2.1.10.: Software – Messung und Bewertung*, pages 32–49. Deutscher Universitätsverlag, Wiesbaden, 2002.

[BLS00] Dirk Beyer, Claus Lewerentz, and Frank Simon. Flattening inheritance structures - or - getting the right picture of large oo-systems. Technical Report I-12/2000, BTU Cottbus, 2000.

[BLS01] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object oriented systems. In R. Dumke and A. Abran, editors, *Proceedings of the 10th International Workshop on Software Measurement (IWSM 2000, Berlin): New Approaches in Software Measurement*, LNCS 2006, pages 1–17. Springer-Verlag, Berlin, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1986.

[Ciu99] Oliver 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.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John 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.

[HHL02] Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In *Proceedings of the 6th International Conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Design and Process Science, 2002.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE 1996)*, pages 208–215. IEEE Computer Society Press, 1996.

[LN03] Claus Lewerentz and Andreas Noack. CrocoCosmos – 3D-visualization of large object-oriented programs. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*. Springer-Verlag, 2003.

[ML96] Jean Mayrand and Bruno Lagu. Object oriented architecture assessment using metrics. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1996)*, 1996.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 338–348, 2002.

[RSK00] Sebastien Robitaille, Reinhard Schauer, and Rudolf K. Keller. Bridging program comprehension tools by design navigation. In *International Conference on Software Maintenance (ICSM 2000)*, pages 22–32. IEEE Computer Society, 2000.

[SL97] Frank Simon and Claus Lewerentz. Integration of an object-oriented metrics tool into SNiFF+. Technical Report I-22/1997, BTU Cottbus, 1997.