Simple and Efficient Relational Querying of Software Structures

Dirk Beyer, Andreas Noack, Claus Lewerentz
Software Systems Engineering Research Group
Brandenburg Technical University at Cottbus, Germany
{db,an,cl}@informatik.tu-cottbus.de

Abstract

Many analyses of software systems can be formalized as relational queries, for example the detection of design patterns, of patterns of problematic design, of code clones, of dead code, and of differences between the as-built and the as-designed architecture. This paper describes the concepts of CrocoPat, a tool for querying and manipulating relations. CrocoPat is easy to use, because of its simple query and manipulation language based on predicate calculus, and its simple file format for relations. CrocoPat is efficient, because it internally represents relations as binary decision diagrams, a data structure that is well-known as a compact representation of large relations in computer-aided verification. CrocoPat is general, because it manipulates not only graphs (i.e. binary relations), but n-ary relations.

1. Introduction

Querying and manipulating graphs or relations has many applications in reverse engineering:

- The importance of the detection of design patterns [17] in understanding and redocumenting object-oriented programs is increasingly recognized. Many tools have been developed (Pat [26], the tool of Antoniol et al. [1], VizzAnalyzer [19]) or extended (SPOOL [25], FUJABA [32]) for this purpose.
- The detection of patterns of problematic design helps in assessing the design quality of programs, and is a first step towards the improvement of the design quality. The tools Hy+[29], Pattern-Lint [33], RPA [16], IAPR [23], Goose [11], and Grok [14] were used to detect structures like cyclic dependencies and irregular inheritance.
- Graph pattern matching was also applied to extract scenarios from models of source code [39].
- The detection of repeated subgraphs supports the identification of code clones [27] and the inductive inference of design patterns [34, 36].
- The forward traversal of call and inheritance graphs is used to detect dead code, and the backward traversal is applied for change impact analysis [10, 16].
- As software systems evolve, the as-built architecture often diverges from the as-designed architecture. The detection of inconsistencies between the as-built and the as-designed architecture, and the transformation of the as-built or the as-designed architecture to achieve consistency support the comprehension and modification of software [33, 16, 30, 14, 31].
- In the reverse engineering of large software systems, multiple views on different abstraction levels are generated. The combination of several views to create new views [24], the lifting and lowering of relations between program entities to create views on higher or lower abstraction levels, and the hiding of parts of a view [16, 14] can be formalized as operations on relations.

A general purpose tool for querying and manipulating relations in reverse engineering should provide a simple but expressive query and manipulation language, and efficiency for large relations. This paper addresses these requirements. In Section 2.2, we introduce a variant of predicate calculus for the manipulation of n-ary relations. It is sufficiently expressive to specify graph patterns of arbitrary size, and thus resolves a problem that was identified as the main loss of (binary) relational algebra by Fahmy, Holt and Cordy [15]. In Section 2.3, we propose to use the data structure binary decision diagram (BDD, [6]) for the internal representation and manipulation of relations. To evaluate these concepts, we implemented them in a tool called CrocoPat, and applied this tool to detect design patterns and design problems in object-oriented software systems. Section 3 reports the results of this evaluation. Finally, Section 4 compares our approach to related work.

Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)
1095-1350/03 $ 17.00 © 2003 IEEE

2. Querying and Manipulating Relations

Querying and manipulating relations requires a database for relations, a query and manipulation language, and data structures and algorithms for the manipulation of relations. The three subsections of this section describe our choices for these parameters.

2.1. Database for Relations

A lightweight calculator for relations should facilitate data exchange with other tools, and it should not require the installation of other programs like database management systems. Reading and writing relations from and to plain text files in Rigi Standard Format (RSF, [38, Section 4.7.1]) fulfills these requirements.

Originally, an RSF file represents binary relations. It is a sequence of triples, one triple on a line. The first element of each triple gives the name of a relation variable, and the second and the third element give the related entities. For example, the following RSF file assigns the value $\{(\texttt{A},\texttt{B}),\ (\texttt{C},\texttt{A})\}$ to the relation variable `CALL` and the value $\{(\texttt{C},\texttt{A})\}$ to the relation variable `INHERIT`:

```
CALL      A         B
CALL      C         A
INHERIT   C         A
```

The RSF is easily generalized from binary relations to n-ary relations ($n \geq 1$) by allowing not only triples, but arbitrary tuples in each line. For each relation, all tuples must have the same number of elements. For example, the following file is *not* allowed:

```
REL       A         B
REL       A         B         C
```

2.2. Query and Manipulation Language

First-order predicate calculus is a well-known, reasonably simple, precise and powerful language. In contrast to the languages used by calculators for *binary* relations (like Grok [20], RPA [16], and RelView [2]), it is sufficiently expressive to specify graph patterns of arbitrary size. Thus predicate calculus, augmented with statements for the output of relations, is a suitable basis of a language for manipulating and querying relations in reverse engineering.

Such a language could be purely declarative, but we decided to give the user the option to control the order of the calculations. For example, the user can explicitly store and reuse a frequently needed intermediate result, to avoid its repeated calculation and to structure the program. However, the user is not required to do so, because the implementation avoids repeated calculations automatically (in most cases) through the use of caches.

Programs of our language are sequences of semicolon-separated *statements*. (Syntactic elements are italicized.) There are only two kinds of *statements*: *assignment statements* and *output statements*. Statements for the input of relations are unnecessary, because input RSF files can be specified as command line parameters and automatically loaded before the execution of the program.

2.2.1 Context-Free Syntax of Assignment Statements

Assignment statements have the form *atomic_expression* `:=` *expression*. The syntax of *expressions* conforms to first-order predicate calculus, with three major exceptions:

- Terms can only be variables or constants, there are no functions.
- There is a special operator for the transitive closure.
- There is a built-in binary relation `=`.

The following grammar specifies the context-free syntax of *expressions* including *atomic expressions*:

expression	::=	`(`*expression*`)`
	|	`TRUE`
	|	`FALSE`
	|	*atomic_expression*
	|	*term* `=` *term*
	|	`!` *expression*
	|	*expression* `^` *expression*
	|	*expression* `+` *expression*
	|	*expression* `->` *expression*
	|	`EX(`*variable*`,` *expression*`)`
	|	`FA(`*variable*`,` *expression*`)`
	|	`TC(`*expression*`,` *variable*`,` *variable*`)`
atomic_expression	::=	*relation_variable* `(`*term_list*`)`
term_list	::=	*term* | *term_list*`,` *term*
term	::=	`"`*constant*`"` | *variable*

Variables and *relation variables* are strings of letters, digits, and the underscore, starting with a letter or the underscore. *Constants* are strings of arbitrary characters excluding double quotes.

2.2.2 Semantics and Context Conditions of Assignment Statements

Like the syntax, the semantics of *expressions* of our language conforms to the semantics of expressions in predicate calculus. So the reader will be able to understand the example programs in Section 3 and to write simple programs without explicit knowledge of the technical details

described in this subsection. The following definition proceeds bottom-up (i.e. from *variables* to *assignment statements*) and is kept concise through references to the well-known semantics of predicate calculus.

The domain of the *variables* is the set of all tuple elements of the input database (except the first tuple elements, which give the names of the *relation variables*) and all *constants* in the program. We call this set DOM. For the example input database

```
CALL      A         B
CALL      C         A
```

we have $DOM = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ (when no additional *constants* appear in the program). The scope of each *variable* is limited to one *statement*, so *variables* with same name are considered different if they appear in different *statements*.

For each *relation variable*, the number of elements of its input RSF tuples (without the first element) and the number of *terms* in its *atomic expressions* have to be equal and determine its arity. The domain of each *relation variable* is the set of all relations over DOM with this arity. For the above input database, the domain of the *relation variable* `CALL` is $2^{\{\texttt{A},\texttt{B},\texttt{C}\}\times\{\texttt{A},\texttt{B},\texttt{C}\}}$ (i.e. the set of all binary relations over $\{\texttt{A}, \texttt{B}, \texttt{C}\}$) and `CALL` can only be used in *atomic expressions* of the form `CALL(`*term*`,` *term*`)`.

The semantics of *atomic expressions* is the same as in predicate calculus. The symbols `!`, `^`, `+`, `->`, `EX`, and `FA` mean negation, conjunction, disjunction, implication, existential quantification, and universal quantification, respectively. The binary relation `=` is predefined as $\{(u, v) \mid u, v \in DOM, u = v\}$. We preferred the more familiar notation *term* `=` *term* to the more consistent `=(`*term*`,` *term*`)`.

To define the context conditions for the transitive closure operator `TC`, we need the notion of a free *variable*. A *variable* is free in an *expression* if it occurs outside the scope of a quantifier with this *variable*. The semantics of `TC(`e`, x, y)` is only defined if the *variables* `x` and `y` are free in the *expression* e, and no other *variables* are free in e. Because e contains exactly two free *variables*, it represents a *binary* relation. The expression `TC(`e`, x, y)` represents the transitive closure of this binary relation.

An *assignment statement* has to fulfill two context conditions. Firstly, all *relation variables* that occur in the right hand side must be defined either in the input database or in the left hand side of an earlier *assignment statement*. Secondly, the set of *variables* in the *term list* of the left hand side must be equal to the set of free *variables* of the right hand side. (So the relations at both sides have the same arity.) The semantics of an *assignment statement* `R(`tl`)` `:=` e (where tl is a *term list* and e is an *expression*) is that a relation is assigned to the *relation variable* `R` such that the *expression* `R(`tl`)` (after the assignment) is equivalent to the *expression* e (before the assignment).

2.2.3 Output Statements

There are two *output statements*:

output_statement ::= `PRINT "`*text*`"`
| `SAVE` *atomic_expression*

The `PRINT` *statement* prints *text* to the standard output. The `SAVE` *statement* writes the relation given by the *atomic expression* into an RSF file whose name can be specified as command line parameter. As in the right hand sides of *assignment statements*, the *relation variable* in the *atomic expression* has to be defined in the input database or as left hand side of an *assignment statement* before it appears in a `SAVE` *statement*.

2.3. Representation of Relations with Binary Decision Diagrams

An important problem in querying graphs and relations is efficiency. For many related problems no polynomial-time algorithm is known. For example, the decision if there is a subgraph of one graph which is isomorphic to another graph is NP-complete [18]. Besides time, memory is also a problem: For a set M with 1000 elements, n-ary relations over M can have 10^{3n} elements.

Experience in computer-aided verification shows that the data structure binary decision diagram (BDD) can represent even huge relations efficiently [8]. BDDs and an associated set of manipulation algorithms were introduced by Bryant [6]. The worst-case time required for the BDD operations is bounded by polynomials of the sizes of the operand BDDs. So when BDDs are small, their manipulation is efficient, even if they represent huge relations. In the following, we will shortly introduce BDDs and give an example how they represent large relations efficiently. For a more detailed introduction to BDDs see e.g. [7].

A BDD is a rooted directed acyclic graph. It has decision nodes, and two terminal nodes called 0-terminal and 1-terminal. Each decision node is labeled by a Boolean variable and has two children called low child and high child. We only use *ordered* BDDs which means that the variables occur in the same order on every path from the root to a terminal node. A BDD is maximally reduced with respect to two rules: Merge any isomorphic subgraphs, and eliminate any node whose two children are isomorphic.

A BDD represents a relation over $\{0, 1\}$, i.e. a set of bit vectors. Relations over other sets than $\{0, 1\}$ can be easily transformed to relations over $\{0, 1\}$ by binary encoding. The bit vectors represented by a BDD correspond to the paths from the root node to the 1-terminal. The vector element that corresponds to a node has the value 0 if the path descends to the low child and the value 1 if the path descends to the high child.

As an example, consider a program with three classes A, B and C, where A calls B and C, B calls A, and C calls A. This can be formalized as a binary relation $CALL$ over $\{A, B, C\}$:

$$CALL = \{(A, B), (A, C), (B, A), (C, A)\}$$

To represent this relation as BDD, it must be transformed into a relation $CALL'$ over $\{0, 1\}$. Encoding A by $(0, 0)$, B by $(0, 1)$, and C by $(1, 0)$ results in:

$$CALL' = \{(0,0,0,1),\ (0,0,1,0),\ (0,1,0,0),\ (1,0,0,0)\}$$

The BDD representation of the relation $CALL'$ (and thus of $CALL$) is shown in the left part of Figure 1. Edges to low children are represented as dotted lines, and edges to the 0-terminal are omitted to avoid clutter. The four bit vectors in the relation $CALL'$ correspond to the four paths from the root node to the 1-terminal in the BDD. For example, the bit vector $(0, 0, 0, 1)$ corresponds to the leftmost path x_1 - dotted line - x_2 - dotted line - x_3 - dotted line - x_4 - solid line - 1-terminal.

An extension of this example shows how BDDs can stay small even for large relations. Consider all chains of k calls, i.e. all tuples of $k+1$ classes $(c_1, c_2, ..., c_{k+1})$ with $(c_i, c_{i+1}) \in CALL$ for all $i \in \{1, ..., k\}$. For $k=1$, the set of these tuples is $CALL = \{(A, B), (A, C), (B, A), (C, A)\}$, for $k=2$ it is $\{(A, B, A), (A, C, A), (B, A, B), (B, A, C), (C, A, B), (C, A, C)\}$, for $k=3$ there are 8 such tuples, etc. In general, there are $2^{(k+3)/2}$ such tuples for odd $k \geq 1$, so the size of the relation grows exponentially with k.

However, the BDD representation grows only linearly with k. To see this, compare the two BDDs in Figure 1. The left BDD represents the relation for $k = 1$, the right BDD for $k = 3$. The increase of k from 1 (by 2) to 3 has added the 10 nodes labeled with x_3, x_4, x_5, and x_6, which are in fact two copies of the subgraph induced by the nodes labeled with x_5 and x_6. Each further increase of k by 1 will again add one copy of this subgraph, i.e. only five nodes.

The simple example illustrates a general problem: The size of intermediate results obtained in searching graph patterns often grows exponentially with the size of the pattern. This problem is well known from joining tables in relational databases. BDDs represent many of these large relations efficiently.

3. Evaluation

In the previous section, we proposed a language and a data structure for querying and manipulating relations. We argued that they have the following benefits:

- The language is sufficiently powerful to express many analyses of software structures.

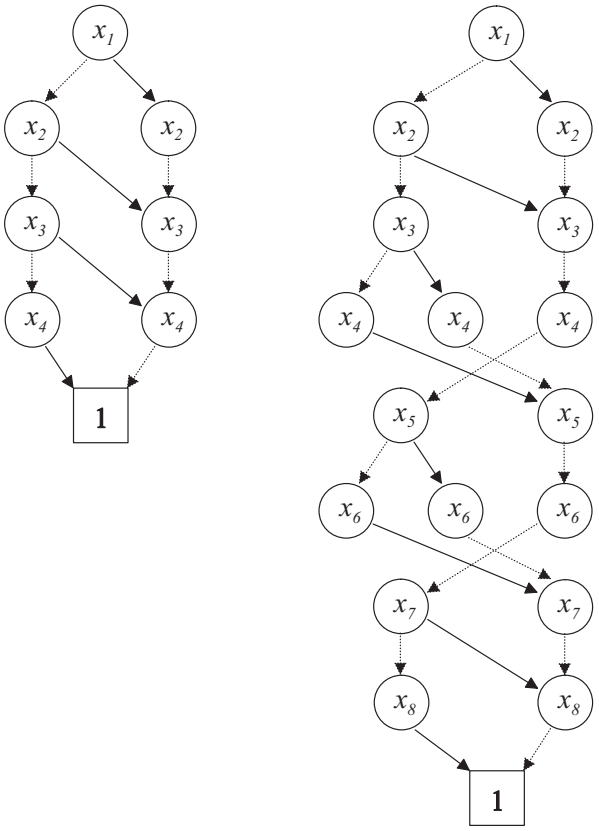

Figure 1. Left: BDD representation of the relation $CALL$; Right: BDD representation of all chains of three $CALL$s

- The programs of the language are reasonably easy to understand and to write.
- The BDD representation of relations scales well to the analysis of large software systems.

To validate the concepts empirically, we implemented them in the tool CrocoPat, and used this tool to analyze object-oriented software systems. For each analysis, we present the program and performance results for CrocoPat, and compare them to the corresponding data for Grok. Grok [20] is a calculator for binary relations that has been applied to many reverse engineering problems (see [15]), and "has been optimized to handle large factbases" [21].

3.1. Method

Evaluation of the language. The analyses performed in our experiments should have proven to be useful in reverse engineering. Therefore we chose analyses that were reported to yield useful results in the reverse engineering lit-

erature. We excluded analyses that are easy to formulate using software measures, like "Classes should not have more than seven attributes.", because our concepts are meant to complement software measurement. For each analysis, we compare the programs for CrocoPat and Grok.

Evaluation of efficiency. The scaling behavior of the BDD representation is best illustrated by contrasting performance data for software systems of different sizes, including large systems. We chose the well-known Java systems JHotDraw 5.2, the AWT of the Java 2 Platform Standard Edition 1.4.0 (JDK 1.4.0 AWT), JWAM 1.6, the complete Java 2 Platform Standard Edition 1.4.0 (JDK 1.4.0), and Eclipse 2.02. Table 1 shows their characteristics. Here LOC is the total number of carriage returns in the source code, and RSF lines is the number of tuples in the extracted RSF file.

System	Classes	LOC	RSF lines
JHotDraw 5.2	168	17 819	878
JDK 1.4.0 AWT	384	141 267	1 504
JWAM 1.6	2 397	284 818	12 298
JDK 1.4.0	5 312	1 179 576	28 699
Eclipse 2.02	8 925	1 181 270	63 121

Table 1. Example systems for performance evaluation

We used the tool SNiFF+ [37] to extract RSF files from the source code of these systems. We extracted the call, containment, and inheritance relations between classes. (Here containment means that a class contains an attribute whose type is another class. Inheritance includes `extends` and `implements` relations.) For example, from the source code

```
class ContainedClass {}
class SuperClass {}
class SubClass extends SuperClass {
    ContainedClass c;
}
```

the following RSF file is extracted:

```
INHERIT   SubClass  SuperClass
CONTAIN   SubClass  ContainedClass
```

For each analysis, we report the computation times of CrocoPat (version 1.2) and Grok (version R15.0). (This publicly available version of Grok does *not* include graph pattern matching as described in [39]. The times for Grok are only given when the analysis could be expressed in Grok's language.) The computation times are given in seconds of processor time on a Linux PC with 1 GHz AMD Athlon processor and 1280 MB memory. For CrocoPat we restricted the memory usage to 12 MB for the first three systems, and to 50 MB for JDK 1.4.0 and Eclipse 2.02. For Grok we report memory overflow (MO) when it uses more than 400 MB.

3.2. Experimental Results

Design pattern. It has often been argued that the knowledge of design pattern instances helps in understanding object-oriented programs. Many tools have been developed or extended for the automatic detection of design patterns instances, e.g. Pat [26], the tool of Antoniol et al. [1], VizzAnalyzer [19]), SPOOL [25], and FUJABA [32].

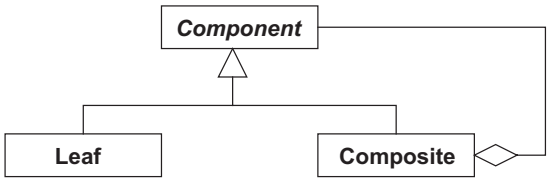

Figure 2. Composite design pattern [17]

Figure 2 shows the class diagram of the Composite design pattern [17]. To identify possible instances of this pattern we compute all triples of a Component class, a Composite class, and a Leaf class, such that (1) the Composite and the Leaf are subclasses of the Component, (2) the Composite contains the Component, and (3) the Leaf does not contain the Component.

```
CompPat(Component, Composite, Leaf) :=
      INHERIT(Composite, Component)
    ^ CONTAIN(Composite, Component)
    ^ INHERIT(Leaf, Component)
    ^ ! CONTAIN(Leaf, Component);
SAVE CompPat(Component, Composite, Leaf);
```

Figure 3. Composite (CrocoPat program)

The translation of these conditions to a CrocoPat program is straightforward, and the resulting program is shown in Figure 3. The relations `INHERIT` and `CONTAIN` are loaded automatically from an RSF file specified as command line parameter. The final statement saves the tuples into an RSF file whose name is also specified as command line parameter. Because the set of all Composite pattern instances is a ternary relation, there is no natural way to describe it in Grok, which is restricted to binary relations.

Table 2 reports the number of detected Composite pattern instances and the computation times of CrocoPat.

System	# Composites	CrocoPat
JHotDraw 5.2	0	0.26
JDK 1.4.0 AWT	15	0.46
JWAM 1.6	21	2.90
JDK 1.4.0	104	24.9
Eclipse 2.02	152	64.7

Table 2. Composite (results)

Cycles. To understand an undocumented class, one has to understand all classes it uses. If one of the (directly or indirectly) used classes is the class itself, understanding it is difficult. Many tools were applied to detect such cyclic structures, e.g. Hy+[29], Pattern-Lint [33], RPA [16], IAPR [23], Goose [11], and Grok [14].

The CrocoPat program in Figure 4 detects cyclic uses of classes, where uses include calls, containment (from the containing to the contained class), and inheritance (from subclass to superclass). In the first statement, the use relation is computed as union of the call, the containment, and the inheritance relation. In the second statement, the transitive closure of the use relation is computed, yielding a relation that also includes all indirect uses. Classes that are related to itself in this transitive closure participate in a cycle of the use relation. The set of these classes is assigned to the relation variable `InCycle` in the second statement, and written to the output RSF file in the third statement.

```
Use(x,y) :=   CALL(x,y)
            + CONTAIN(x,y)
            + INHERIT(x,y);
InCycle(x) := EX(y, TC(Use(x,y),x,y) ^ (x = y));
SAVE InCycle(x);
```

Figure 4. Classes in cycles (CrocoPat program)

Figure 5 shows the corresponding Grok program. The first statement loads the relations from the RSF file. `Use+` is the transitive closure of `Use`, `dom` computes the domain (i.e. the set of all first tuple elements) of a given relation, `id` is the identity relation over a given set, and `ENT` is the set of all entities occurring in the input RSF file.

```
getdb $1
Use := CALL + CONTAIN + INHERIT
InCycle := dom (Use+ ^ id ENT)
putset InCycle $2
q
```

Figure 5. Classes in cycles (Grok program)

Table 3 reports the results of applying these programs to the example systems. The column CIC shows how many classes participate in cycles.

System	CIC	Grok	CrocoPat
JHotDraw 5.2	16	0.21	0.27
JDK 1.4.0 AWT	120	0.44	0.46
JWAM 1.6	38	1.20	2.43
JDK 1.4.0	1304	115	8.75
Eclipse 2.02	2465	MO	38.5

Table 3. Classes in cycles (results)

It is very tedious for a human analyst to find the actual cycles in a list of hundreds of classes which are part of a cycle. With CrocoPat it is possible to compute all cycles of some fixed length. Our program detects the cycles in the order of ascending length. After the detection of all cycles of length k, all edges (except inheritance edges, which cannot be cyclic) that participate in these cycles are deleted. This ensures that the computed cycles of lengths greater than k are not just chains of shorter cycles. The CrocoPat program is simple but rather long and repetitive, so we omit it. Table 4 reports the numbers of cycles of the lengths 2 to 5.

Because the set of all cycles of length n is an n-ary relation, it cannot be computed with the binary relational algebra of Grok for $n > 2$.

System	2	3	4	5	CrocoPat
JHotDraw 5.2	10	0	0	0	1.49
JDK 1.4.0 AWT	61	5	3	0	2.05
JWAM 1.6	14	2	1	0	8.34
JDK 1.4.0	389	81	27	3	70.4
Eclipse 2.02	920	262	41	29	219

Table 4. Cycles of fixed length (results)

Similar classes. Many approaches to code clone detection focus on the lexical or syntactic level (see [9] for an overview). These algorithms can be complemented by the detection of similar classes at the design level.

Figure 6 shows a Grok program that detects all pairs of classes that call the same classes, contain instances of the same classes, and inherit from the same classes. The term `(ENT X ENT) - R` is used to compute the complement of the relation `R`, `*` is the composition operator, and `inv` swaps the first and the second element of all tuples of a relation. Figure 7 shows the corresponding CrocoPat program.

```
getdb $1

Rleft1 := CALL * ((ENT X ENT) - inv CALL)
Rleft2 := ((ENT X ENT) - CALL) * inv CALL
Rleft  := Rleft1 + Rleft2
IdCall := (ENT X ENT) - Rleft

Rleft1 := CONTAIN * ((ENT X ENT) - inv CONTAIN)
Rleft2 := ((ENT X ENT) - CONTAIN) * inv CONTAIN
Rleft  := Rleft1 + Rleft2
IdCont := (ENT X ENT) - Rleft

Rleft1 := INHERIT * ((ENT X ENT) - inv INHERIT)
Rleft2 := ((ENT X ENT) - INHERIT) * inv INHERIT
Rleft  := Rleft1 + Rleft2
IdInh  := (ENT X ENT) - Rleft

Ident  := IdCall ^ IdCont ^ IdInh
appendRelToFile Ident $2
q
```

Figure 6. Similar classes (Grok program)

```
IdCall(x,z):=
    ! (  EX(y, CALL(x,y) ^ (! CALL(z,y)))
       + EX(y, (! CALL(x,y)) ^ CALL(z,y)));

IdCont(x,z):=
    ! (  EX(y, CONTAIN(x,y) ^ (! CONTAIN(z,y)))
       + EX(y, (! CONTAIN(x,y)) ^ CONTAIN(z,y)));

IdInh(x,z):=
    ! (  EX(y, INHERIT(x,y) ^ (! INHERIT(z,y)))
       + EX(y, (! INHERIT(x,y)) ^ INHERIT(z,y)));

Ident(x,z) := IdCall(x,z)^IdCont(x,z)^IdInh(x,z);
SAVE Ident(x,z);
```

Figure 7. Similar classes (CrocoPat program)

We feel that both programs are unnecessarily difficult to understand and to develop. Intuitively, two classes are similar if all classes that are called by the first class are also called by the second class, and all classes that are called by the second are also called by the first, and analogous conditions hold for containment and inheritance. With the "for all" quantifier of predicate calculus we can express these conditions directly. Figure 8 shows the corresponding CrocoPat program. In fact, this program was our first formalization of the similar classes, and we developed the programs in the Figures 6 and 7 only because we had to use the composition operator `*` instead of quantifiers in Grok.

```
IdCall(x,z) :=
      FA(y,  (CALL(x,y) -> CALL(z,y))
           ^ (CALL(z,y) -> CALL(x,y)));

IdCont(x,z) :=
      FA(y,  (CONTAIN(x,y) -> CONTAIN(z,y))
           ^ (CONTAIN(z,y) -> CONTAIN(x,y)));

IdInh(x,z) :=
      FA(y,  (INHERIT(x,y) -> INHERIT(z,y))
           ^ (INHERIT(z,y) -> INHERIT(x,y)));

Ident(x,z) := IdCall(x,z)^IdCont(x,z)^IdInh(x,z);
SAVE Ident(x,z);
```

Figure 8. Similar classes (CrocoPat program with `FA` operator)

Table 5 reports not the number of pairs of similar classes, but the number of different classes in these pairs. The set of classes in similar pairs was obtained through a postprocessing step. We preferred the number of similar classes to the number of pairs of similar classes because the latter is large and hard to interpret: For k classes that are pair-wise similar, the number of pairs is k^2.

Degenerate inheritance. When a class inherits from another class directly and indirectly, the direct inheritance is probably redundant or even misleading. In a measurement-based quality assessment of the JWAM framework (see [5] for an overview), one of the restructuring recommendations was to avoid such inheritance structures. In this original assessment, one instance of the pattern was found indirectly. Our structural analyses reveal that there exist much more instances.

System	# sim. cls.	Grok	CrocoPat
JHotDraw 5.2	59	1.54	0.29
JDK 1.4.0 AWT	139	5.72	0.42
JWAM 1.6	481	MO	2.61
JDK 1.4.0	1 663	MO	7.34
Eclipse 2.02	1 784	MO	18.1

Table 5. Similar classes (results)

A degenerate inheritance structure consists of three classes, where the first and the second class are direct superclasses of the third class, and the first class is a (not necessarily direct) superclass of the second class. Figure 9 shows the straightforward CrocoPat program for this pattern. Again, there is no natural way to describe the pattern in Grok, because it is restricted to binary relations.

```
DegInh(a,b,c) :=   INHERIT(c,b)
                 ^ INHERIT(c,a)
                 ^ TC(INHERIT(b,a), b, a);
SAVE DegInh(a,b,c);
```

Figure 9. Degenerate inheritance (CrocoPat program)

Table 6 reports the number of detected pattern instances and the computation times of CrocoPat.

System	# degenerate triples	CrocoPat
JHotDraw 5.2	1	0.23
JDK 1.4.0 AWT	4	0.32
JWAM 1.6	161	2.60
JDK 1.4.0	1 233	52.3
Eclipse 2.02	334	38.9

Table 6. Degenerate inheritance (results)

Subclass knowledge. Superclasses should not know their subclasses, because superclasses should be understandable and reusable independently of their subclasses, and modifying subclasses should not affect the superclass. Subclass knowledge is a special case of the cyclic usage structures discussed earlier, and was detected e.g. with the tools Pattern-Lint [33] and Goose [11].

A basic version of this pattern is a pair of classes, such that the second class is a (not necessarily direct) subclass of the first class and the first class (possibly indirectly) calls or contains the second class. The corresponding Grok and CrocoPat programs are shown in Figure 10 and Figure 11, respectively.

Table 7 reports the number of detected instances of subclass knowledge and the computation times of Grok and CrocoPat.

```
getdb $1
Know := (CALL + CONTAIN)+ ^ (inv INHERIT)+
appendRelToFile Know $2
q
```

Figure 10. Subclass knowledge (Grok)

```
Know(super,sub) :=
      TC(CALL(super,sub) | CONTAIN(super,sub),
         super, sub)
    & TC(INHERIT(sub,super), sub, super);
SAVE Know(super,sub);
```

Figure 11. Subclass knowledge (CrocoPat)

System	# instances	Grok	CrocoPat
JHotDraw 5.2	0	0.21	0.19
JDK 1.4.0 AWT	74	0.43	0.45
JWAM 1.6	11	1.10	2.74
JDK 1.4.0	2720	75.6	24.8
Eclipse 2.02	2295	MO	83.9

Table 7. Subclass knowledge (results)

3.3. Discussion

Three of the six analyses could be naturally expressed in Grok and CrocoPat, because the results were unary or binary relations. In two of these cases (classes in cycles and subclass knowledge), the programs of Grok and of CrocoPat were quite similar, and it is probably a matter of taste which to prefer. Compared to Grok, CrocoPat has the quantifiers as additional syntactic elements, which are well-known from predicate calculus and add much expressiveness. Conversely, we had to use several syntactic elements of Grok that are unnecessary in CrocoPat (`*`, `inv`, `dom`, `ENT`). In one case (similar classes), we felt that the availability of quantifiers in CrocoPat made the development and understanding of the program much easier.

As noted in [15], graph patterns with more than two nodes generally cannot be expressed in binary relational algebra. The examples of the Composite design pattern and degenerate inheritance show that such patterns can be specified easily in the language of CrocoPat.

Concerning efficiency, no analyses with CrocoPat required more than four minutes of time and 50 MB of memory. Grok outperformed CrocoPat for some small problems, but CrocoPat scaled much better to large systems. In particular, Grok could not complete any analysis of Eclipse because it required too much memory, and took much more time for all analyses of JDK.

It is worth noting that the greater generality of CrocoPat (compared to Grok) does not lead to an increased need for resources. We could not compare the performance for relations of arity greater than two. However, as indicated in Section 2.3, we expect that the full superiority of BDDs over other data structures shows only for these relations.

3.4. Comparison with SQL

Relational databases and SQL are used in several reverse engineering toolsets for querying and manipulating relations (e.g. in the Dali workbench [24], CppSpec [35], Sotograph [3]). In this subsection we briefly compare the performance of the database management system MySQL 3.23.48 and CrocoPat in the computation of transitive closures, which appear frequently in structural analyses.

We computed the transitive closure of the `USE` relation, i.e. the union of `CALL`, `CONTAIN`, and `INHERIT`. The CrocoPat program is simply

```
Closure(x,y) := TC(USE(x,y),x,y);
```

Because the transitive closure is not directly expressible in SQL, we developed the SQL script in Figure 12, and executed it repeatedly until the fixed point was reached. To improve the performance of MySQL, we used heap tables (i.e. tables stored in main memory, not on hard disc), `INT(4)` as type of all table columns, and indexes where appropriate.

The computation times are shown in Table 8. For the last two systems, MySQL needs more than 400 MB of memory. When normal tables instead of heap tables are used, the computation time explodes and we interrupted the computations after one hour. Even if other database management systems were an order of magnitude faster, the performance would still not be satisfactory.

```
INSERT INTO ClosureNew (x,y)
  SELECT * FROM Closure;
INSERT INTO ClosureNew (x,y)
  SELECT l.x, r.y
  FROM Closure l, Closure r WHERE l.y = r.x;
DELETE FROM Closure;

INSERT INTO Closure (x,y)
  SELECT DISTINCT * FROM ClosureNew;
DELETE FROM ClosureNew;
```

Figure 12. Transitive Closure (SQL)

System	MySQL	CrocoPat
JHotDraw 5.2	0.35	0.18
JDK 1.4.0 AWT	29.5	0.27
JWAM 1.6	16.8	2.24
JDK 1.4.0	MO	8.37
Eclipse 2.02	MO	37.7

Table 8. Transitive closure (results)

4. Related Work

SQL is a well-known language for querying and manipulating relations. The lack of a transitive closure operator and the insufficient performance of relational databases for large graphs were already discussed in Section 3.4. Furthermore, CrocoPat facilitates data exchange because of its simple file format for relations, and is easier to install and maintain than relational database management systems.

The logic programming language Prolog [12] is quite similar to the language of CrocoPat. Prolog has been used to detect design patterns and design problems in the tools Pat [26], Pattern-Lint [33] and Goose [11]. CrocoPat differs from Prolog interpreters in that it is tailored for the use in reverse engineering: It has a much smaller language, efficient algorithms that are optimized for this reduced language, and it uses a standard file format of reverse engineering.

Calculators for binary relational algebra that have been used in reverse engineering include Grok [20], RPA [16], and RelView [2]. However, there are some practically important queries that cannot be expressed with binary relations. A graph pattern with n nodes, for example, is an n-tuple. Grok was extended to support graph pattern matching [39], but this resulted in a more complex language that still does not support other operations on n-ary relations.

The program understanding toolset GUPRO [13] provides the textual graph querying language GReQL [28]. The approach differs from CrocoPat in several respects: GReQL focusses on querying, while CrocoPat can also create and modify relations. GReQL focusses on graphs (binary relations), CrocoPat on n-ary relations. GReQL requires the specification of a graph class, while CrocoPat can manipulate directed, attributed graphs without such a specification. Visual graph querying languages include GraphLog in the tool Hy+ [29], annotated graphs in IAPR [23], and a subset of UML in FUJABA [32].

The graph rewriting rule based specification and rapid prototyping language PROGRES [4] has a purely textual and a combination of visual and textual notation. It is expressive, but also much more complicated than CrocoPat.

As input and output format for relations CrocoPat uses the tuple notation of the Rigi Standard Format (RSF, [38, Section 4.7.1]), which facilitates data exchange with other tools. CrocoPat does not yet support the XML-based Graph Exchange Language (GXL, [22]) because the additional features of GXL are not needed.

Querying graphs and relations is related to NP-hard problems like subgraph isomorphism, and therefore efficiency is a central problem. Binary decision diagrams are successfully applied in computer-aided verification for the efficient representation and manipulation of huge relations (see e.g. [8]). However, no BDD-based calculator for relations was available in reverse engineering (with the exception of RelView [2], which is limited to binary relations where the potential of BDDs is not fully exploited). The experimental results in Section 3 confirm the excellent performance of our BDD-based implementation.

5. Conclusion

Querying and manipulating relations has many applications in reverse engineering. However, existing tools for calculating with relations are not powerful enough to detect graph patterns with more than two nodes, or difficult to use and integrate with other tools, or inefficient for some practically important operations.

We proposed to use the well-known language of predicate calculus for the manipulation, and the data structure BDD for the efficient internal representation of n-ary relations. We implemented these concepts in the tool CrocoPat, and evaluated this tool in structural analyses of five object-oriented software systems. The experiments confirmed that the language of CrocoPat is sufficiently expressive and reasonably easy to use, and that CrocoPat scales well to the analysis of large software systems.

Recently, CrocoPat was integrated into the commercial software analysis and visualization workbench Sotograph [3]. Sotograph is based on relational databases, and its users demanded CrocoPat for efficient graph pattern matching. Through the use of RSF as input and output format the integration was easy and inexpensive.

The tool CrocoPat is publicly available via Internet from http://www.software-systemtechnik.de/CrocoPat.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th IEEE International Workshop on Program Understanding (IWPC 1998)*, pages 153–160, 1998.

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

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

[4] D. Blostein and A. Schürr. Computing with graphs and graph transformations. *Software: Practice & Experience*, 29(3):197–217, 1999.

[5] H. Breitling, C. Lewerentz, C. Lilienthal, M. Lippert, F. Simon, and F. 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.

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

[7] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[9] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 36–43, 2002.

[10] Y.-F. Chen, E. R. Gansner, and E. Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions On Software Engineering*, 24(9):682–694, 1998.

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

[12] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 3rd edition, 1987.

[13] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO – generic understanding of programs. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[14] H. Fahmy and R. C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 88–96, 2000.

[15] H. Fahmy, R. C. Holt, and J. R. Cordy. Wins and losses of algebraic transformations of software architectures. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 51–60, 2001.

[16] L. M. G. Feijs, R. L. Krikhaar, and R. C. van Ommering. A relational approach to support software architecture analysis. *Software: Practice & Experience*, 28(4):371–400, 1998.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[19] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pages 94–103, 2003.

[20] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE 1998)*, pages 210–219, 1998.

[21] R. C. Holt. Introduction to the Grok language, 2002. http://plg.uwaterloo.ca/ holt/papers/grok-intro.html.

[22] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 162–171, 2000.

[23] R. Kazman and M. Burth. Assessing architectural complexity. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR 1998)*, pages 104–112, 1998.

[24] R. Kazman and S. J. Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse (ICSR 1998)*, pages 290–299, 1998.

[25] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 226–235. ACM, 1999.

[26] C. Krämer and L. 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, 1996.

[27] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 301–309, 2001.

[28] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering (CSMR 1999)*, pages 42–50, 1999.

[29] A. O. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software – Concepts & Tools*, 16(4):170–182, 1995.

[30] K. Mens and R. Wuyts. Declarative codifying software architectures using virtual software classifications. In *Proceedings of Technology of Object-Oriented Languages and Systems Europe 1999*, pages 33–45, 1999.

[31] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions On Software Engineering*, 27(4):364–380, 2001.

[32] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 338–348, 2002.

[33] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE 1996)*, pages 387–396, 1996.

[34] F. Shull, W. L. Melo, and V. R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, Computer Science Department, University of Maryland, 1996.

[35] H. M. Sneed and T. Dombovari. Comprehending a complex, distributed, object-oriented software system: A report from the field. In *Proceedings of the 7th International Workshop on Program Understanding (IWPC 1999)*, pages 218–225, 1999.

[36] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*, pages 230–238, 1999.

[37] Wind River Systems, Inc. Sniff+. http://www.windriver.com/products/sniff_plus/.

[38] K. Wong. *Rigi User's Manual, Version 5.4.4*, 1998. http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/.

[39] J. Wu, A. E. Hassan, and R. C. Holt. Using graph patterns to extract scenarios. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 239–247, 2002.