Efficient Relational Calculation for Software Analysis

Dirk Beyer, *Member*, *IEEE*, Andreas Noack, *Member*, *IEEE Computer Society*, and Claus Lewerentz

Abstract—Calculating with graphs and relations has many applications in the analysis of software systems, for example, the detection of design patterns or patterns of problematic design and the computation of design metrics. These applications require an expressive query language, in particular, for the detection of graph patterns, and an efficient evaluation of the queries even for large graphs. In this paper, we introduce RML, a simple language for querying and manipulating relations based on predicate calculus, and CrocoPat, an interpreter for RML programs. RML is general because it enables the manipulation not only of graphs (i.e., binary relations), but of relations of arbitrary arity. CrocoPat executes RML programs efficiently 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. We evaluate RML by giving example programs for several software analyses and CrocoPat by comparing its performance with calculators for binary relations, a Prolog system, and a relational database management system.

Index Terms—Logic programming, graph algorithms, data structures, reverse engineering, reengineering.

1 Introduction

Graphs are widely used as models of software systems. The mapping of software artifacts and their relationships to graph nodes and graph arcs enables the formalization of structural software properties as graph properties and the use of graph algorithms to analyze software systems.

Following this approach, we formalized patterns of proven or problematic design as graph patterns and explored the use of several popular graph analysis tools to detect these patterns. However, none of the evaluated tools succeeded for real-world software systems through either lack of generality or lack of efficiency.

The detection of patterns in graphs requires *generality* because it involves relations of arbitrary arity. Instances of a pattern with n participating software artifacts are naturally modeled as n-tuples, thus sets of such pattern instances are n-ary relations. So, it is difficult to perform these practically important graph analyses with tools that are limited to binary relations, like Grok [1], RPA [2], and RelView [3].

Efficiency in terms of runtime and memory is necessary because graph models of software systems can have thousands of nodes and arcs. However, efficiency is difficult to achieve because finding patterns in graphs involves problems for which no algorithm with polynomial worst-case time complexity is known. This does not preclude the existence of heuristics that are fast enough for many practical problem instances. But, our experiments (described in Section 5) show that widely used systems with expressive languages, in particular MySQL [4] as a popular relational database management system [5] and Quintus Prolog [6] as a leading implementation of the logic programming language Prolog [7], are prohibitively inefficient for practical software analyses of this type.

This article presents concepts for calculating with graphs and relations in the context of software analysis with a focus on the requirements of generality and efficiency. In terms of our experiences with existing tools, our goals were a language that combines Prolog's elegant syntax for manipulating n-ary relations with simplicity and a tool implementation that combines RelView's efficiency with the power to manipulate relations of any arity.

The structure of the article follows from these objectives: Sections 2 and 3 introduce and demonstrate the language, and Sections 4 and 5 describe and evaluate the implementation. The relational expressions of our language RML (Relation Manipulation Language) are based on predicate calculus. Its expressiveness and ease of use is evaluated by giving example programs for several design analyses of object-oriented programs, some of which cannot be naturally expressed with calculators for binary relations. Our tool implementation, CrocoPat, represents relations as binary decision diagrams (BDDs [8]), a data structure that has been successfully used in computer-aided verification for the efficient representation and manipulation of large relations. The efficiency is evaluated by comparing its computation times for two of the most frequent and expensive operations (namely, the computation of transitive closures and the detection of graph patterns) with Grok, RelView, MySQL, and Quintus Prolog.

- *D. Beyer is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720-1770. E-mail: beyer@eecs.berkeley.edu.*
- *A. Noack and C. Lewerentz are with the Software Systems Engineering Research Group, Brandenburg University of Technology, PO Box 101344, D-03013 Cottbus, Germany. E-mail: an@informatik.tu-cottbus.de, cl@tu-cottbus.de.*

Manuscript received 16 Apr. 2004; revised 26 Aug. 2004; accepted 1 Dec. 2004; published online 23 Feb. 2005.
Recommended for acceptance by E. Stroulia and A. van Deursen.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0074-0404.

2 Language

The main goals in the design of the Relation Manipulation Language (RML), a query and manipulation language for relations, were expressiveness and ease of use. RML allows the manipulation of relations of any arity and is thus sufficiently expressive to specify graph patterns of arbitrary size. For ease of use, RML is small, and its elements are well-known from predicate calculus and common imperative programming languages.

This section introduces the relational expressions, numerical expressions, and statements of RML with the help of examples. A complete and formal definition of the RML syntax and semantics is given in [9].

2.1 Relational Expressions

A relational expression is an expression whose result is a relation. A relation of arity n is a set of tuples, where each tuple has n elements. Binary relations, i.e., relations of arity 2, are also called (directed) graphs. In RML, all tuple elements are strings, and relations always contain a finite number of tuples.

The relational expressions in RML are similar to the expressions in first-order predicate calculus [10]. In addition to predicate calculus, RML provides some shortcuts, e.g., for comparing relations and for transitive closures, and some predefined relations, e.g., for comparing strings. On the other hand, RML does not include the functions of predicate calculus (which would map strings to strings) because calculating with strings is not essential for RML's main purpose of calculating with *relations* over strings.

In the examples of this section, we use a binary relation `Call` that contains the three pairs (`A`, `B`), (`A`, `C`), and (`B`, `A`). This relation can be considered as a model of a program with three functions `A`, `B`, and `C`, where `A` calls `B` and `C`, and `B` calls `A`.

Because there exists only one tuple () with 0 elements, there are two relations of arity 0: the set {()} that contains this tuple and the empty set {}. Their names in RML are, respectively, `TRUE()` and `FALSE()`.

Basic relational expressions have the form

rel_variable`(`*term_list*`)`

A *term_list* is a list of comma-separated *term*s. A *term* is either a string literal (delimited by double quotes) or an *attribute*. For example, `Call("A","C")` evaluates to `TRUE()` because the pair of strings (`A`, `C`) is an element of the relation `Call`. The assignment statement `CalledByB(x) := Call("B",x)` with the attribute `x` assigns to the unary relation variable `CalledByB` the set of all functions that are called by `B`, namely, the set {`A`}. The scope of an attribute is a single statement.

RML provides operators for the complement, the intersection, and the union of relations:

`!` *rel_expression*
rel_expression `&` *rel_expression*
rel_expression `|` *rel_expression*

Attributes can be existentially and universally quantified:

`EX(`*attribute*`,` *rel_expression*`)`
`FA(`*attribute*`,` *rel_expression*`)`

For example, the statement `Caller(x) := EX(y, Call(x, y))` assigns to the relation variable `Caller` the set {`A`, `B`} of all functions that call at least one function. The statement `Uncalled(y) := FA(x, ! Call(x,y))` assigns to `Uncalled` the set of all functions that are not called by any function, which is empty here.

The `TC` operator computes the transitive closure of a binary relation:

`TC(`*rel_expression*`)`

The statement `TCall(x,y) := TC(Call(x,y))` assigns to `TCall` all pairs of functions where the first function directly or indirectly (via other functions) calls the second function. Thus, `TCall` contains the six pairs (`A`, `A`), (`A`, `B`), (`A`, `C`), (`B`, `A`), (`B`, `B`), (`B`, `C`).

Relations can be compared with

rel_expression ~ *rel_expression*

where ~ can be = (equality), `!=` (inequality), <= (subset), < (strict subset), >= (superset), or > (strict superset), and the result is either `FALSE()` or `TRUE()`.

RML includes the six predefined binary relations =, `!=`, <=, <, >=, and > for the lexicographical order of strings. So, `SmallerThanC(x) := <(x,"C")` assigns to the relation variable `SmallerThanC` the set of {`A`, `B`} of strings which are lexicographically smaller than `C`.

2.2 Numerical Expressions

A numerical expression is an expression whose result is a number. All numbers in RML are floating-point numbers. The numerical expressions are not specific to RML, but nevertheless useful in some applications and, therefore, briefly listed here.

The basic numerical expressions in RML are numerical literals like `1.23` and numerical variables.

The main purpose of numerical expressions is to calculate with the cardinality (number of elements) of relations. The syntax of the cardinality operator is

`#(`*rel_expression*`)`

RML provides the familiar numerical operators + (addition), - (subtraction), * (multiplication), / (real division), `DIV` (truncating division), and `MOD` (modulo). Numbers can be compared with =, `!=`, <=, <, >=, or >, and the result is either `FALSE()` or `TRUE()`.

2.3 Statements

RML programs are sequences of semicolon-separated statements. There are three kinds of statements: assignments, control structures, and output statements. Statements for the input of relations are unnecessary because input relations are read from standard input before the execution of the program.

The syntax of the assignment statements is

rel_variable`(`*term_list*`)` `:=` *rel_expression*
num_variable `:=` *num_expression*

Examples for assignments were given in the Section 2.1.

The syntax of the control structures is similar to well-known imperative programming languages:

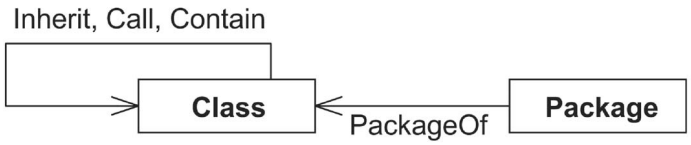

Fig. 1. Simple metamodel of object-oriented software.

`IF` *rel_expression* {*statement_seq*} `ELSE` {*statement_seq*}
`WHILE` *rel_expression* {*statement_seq*}
`FOR` *string_variable* `IN` *rel_expression* {*statement_seq*}

The condition in the `IF` and `WHILE` statement is fulfilled if the relational expression evaluates to `TRUE()`. The `FOR` statement executes its contained statement sequence for each string in the relation, which must be unary.

Finally, the `PRINT` statement prints string literals and results of relational and numerical expressions to the standard output.

2.4 Discussion

2.4.1 Binary versus n-ary Relations

All n-ary relations over a finite universe can be represented by binary and even by unary relations over a finite universe. For example, for an arity n and a finite universe U, the tuples from U^n can be injectively mapped to integers from $\{1, \ldots, |U|^n\}$ and, thus, n-ary relations over U can be mapped to sets of integers.

Nevertheless, it is often more appropriate to represent data as a relation of arity greater than 1 or 2. For example, the obvious representation of a cycle of length 3 in the relation `Call` is a triple, and the obvious representation of a set of such cycles is a ternary relation `Cycle3(x,y,z)`. Using this representation, we can easily query, e.g., for functions that participate in a cycle with `EX(y, EX(z, Cycle3(x, y,z)))` or for cycles which have `A` as the first element with `Cycle3("A",y,z)`. The set of cycles could also be represented as a set of integers, but this representation makes formulating such queries much more difficult and is not interpretable by humans without decoding into triples.

Examples in Section 3 and in [11] show that relations of arity greater than two are not only more natural representations for artificial problems, but also for practical software analyses.

2.4.2 The Universe

In RML, all tuple elements are strings. But, not all strings may be tuple elements, at least for a given RML program and given input data. The set of strings that may be tuple elements is called the *universe*. It is important to define the universe because the result of the complement operator depends on it: The complement of a unary relation `R` is the unary relation of all strings in the universe that are not in `R`.

We identified two restrictions for the universe. First, it should be constant during the execution of an RML program; otherwise, the same relation has different complements at different times. Second, it must be finite because RML is restricted to finite relations to be efficiently interpretable. A possible definition that fulfills these restrictions is that the universe in a given RML program execution contains all tuple elements of the input relations and all string literals that appear on the left-hand side of assignments in the RML program.

The finiteness and immutability of the universe are sometimes inconvenient for the developer of RML programs. Ways to circumvent these limitations are proposed in [9].

3 Applications

To validate the expressiveness and ease of use of RML, this section shows RML programs for several analyses of object-oriented designs. We focus on analyses that have been reported to be useful in the literature: the detection of design pattern instances, the detection of potential design problems, the calculation of design metrics, and the abstraction of design models. Three of the analyses in this section result in relations of arity greater than two and, thus, could not be naturally expressed with binary relational algebra. Before the example programs, the first subsection shortly introduces our underlying metamodel of object-oriented software systems. The section concludes with a list of further potential applications of RML.

3.1 A Simple Metamodel of Object-Oriented Software

A structural model of an object-oriented software system describes the system's entities and the relationships between the entities. Graphs can capture entities and their relationships in a straightforward way: Entities are represented by nodes and relationships by arcs.

For the examples in this section, we use the simple metamodel shown in Fig. 1. It distinguishes only two types of entities: classes and packages. Packages are high-level design elements that can be considered as sets of classes. The unary relation `Class` represents all classes, and the unary relation `Package` represents all packages. The binary relation `PackageOf` relates each package to the classes that it contains.

The models comprise three binary relations for relationships between classes:

- `Inherit`, which relates each subclass to its direct superclass(es);
- `Call`, which relates a class to another class if a method of the first class calls a method of the second class; and
- `Contain`, which relates each class to the types of its attributes.

These relations can be automatically extracted from source code in popular object-oriented programming languages like C++ and Java.

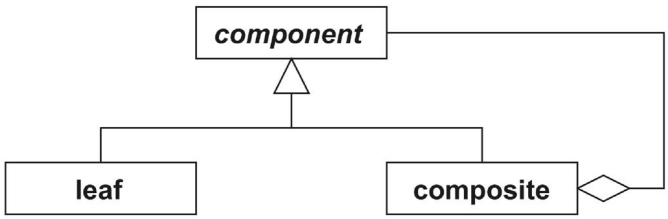

Fig. 2. Class diagram of the Composite design pattern [17].

```
CompPat(component, composite, leaf) :=
                    Inherit(composite, component)
                  & Contain(composite, component)
                  & Inherit(leaf, component)
                  & ! Contain(leaf, component);
PRINT CompPat(component, composite, leaf);
```

Fig. 3. RML program for the detection of the Composite design pattern.

```
Use(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
InCycle(x) := EX(y, TC(Use(x,y)) & =(x,y));
PRINT InCycle(x);
```

Fig. 4. RML program for the detection of classes in cycles.

```
Use(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
Cycle3(x,y,z) := Use(x,y) & Use(y,z) & Use(z,x);
Cycle3(x,y,z) := Cycle3(x,y,z) & <=(x,y) & <=(x,z);
PRINT Cycle3(x,y,z);
```

Fig. 5. RML program for the detection of cycles of three classes.

3.2 Detection of Design Pattern Instances

The knowledge of design pattern instances enables a more abstract description of an object-oriented design, and helps to uncover design rationale. Many tools have been developed or extended for the automatic detection of design patterns instances, e.g., Pat [12], the tool of Antoniol et al. [13], VizzAnalyzer [14], SPOOL [15], and FUJABA [16].

Fig. 2 shows the class diagram of the Composite design pattern [17]. We consider a triple of a component class, a composite class, and a leaf class as a possible instance of this pattern if 1) the composite class and the leaf class are subclasses of the component, 2) the composite class contains the component, and 3) the leaf does not contain the component. The translation of these conditions to the RML program in Fig. 3 is straightforward.

In general, design patterns are hierarchies of subpatterns. For example, the Composite design pattern contains an aggregation relationship from the composite class to the component class. The implementation of this relationship in an object-oriented program can be nontrivial, e.g., involving a container class. So, aggregation is itself a pattern whose instances have to be detected in lower-level information extracted from the source code. In RML, patterns can be easily combined to larger patterns. For example, the relation `Contain` used in the pattern definition in Fig. 3 can be an input relation, but also a set of subpattern instances computed by earlier RML statements.

3.3 Detection of Design Problems

3.3.1 Cyclic References

A group of classes with cyclic references can only be understood and reused as a whole and is hard to modify because changes are likely to propagate through the group. Many tools were applied to detect cyclic usage structures, e.g., Hy+ [18], Pattern-Lint [19], RPA [2], IAPR [20], Goose [21], and Grok [22].

The RML program in Fig. 4 detects cyclic uses of classes, where the uses include calls, containment, and inheritance. In the first statement, the use relation is computed as the 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.

In many large software systems, hundreds of classes participate in cycles, and it is very tedious for a human analyst to find the actual cycles in the list of these classes. In our experience, it is more useful to detect cycles in the order of ascending length. As a part of such a program, the statements in Fig. 5 detect all cycles of three classes.

To see the purpose of the third statement, consider three classes `A`, `B`, and `C` that form a cycle. After the second statement, the relation variable `Cycle3` contains three representatives of this cycle: (`A`, `B`, `C`), (`B`, `C`, `A`), and (`C`, `A`, `B`). The third statement removes two of these representatives from `Cycle3` and keeps only the tuple with the lexicographically smallest class at the first position, namely, (`A`, `B`, `C`).

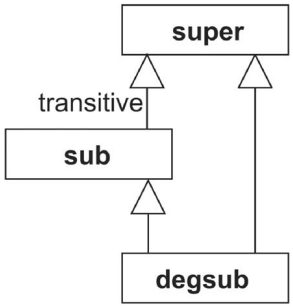

Fig. 6. Class diagram of degenerate inheritance.

3.3.2 Degenerate Inheritance

When a class inherits from another class directly and indirectly, the direct inheritance is redundant and probably even misleading. For example, the subclass may inherit implementations of abstract methods of the superclass from other classes and, thus, does not need to implement these methods itself. As shown in Fig. 6, we model a degenerate inheritance structure as triple of 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. Fig. 7 shows the straightforward RML program for this pattern.

3.3.3 Subclass Knowledge

Superclasses should not know their subclasses because superclasses should be understandable and reusable independently of their subclasses, and modifications of subclasses should not affect the superclass. Subclass knowledge is a special case of the cyclic usage structures discussed earlier and was also detected, e.g., with the tools Pattern-Lint [19] and Goose [21].

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 RML program is shown in Fig. 8.

3.4 Calculation of Design Metrics

Design metrics are essential in the assessment of design quality [23], [24]. Mens and Lanza argue that a large number of object-oriented design metrics can be defined in an unambiguous, simple, and language independent way based on a graph representation of software [25]. Using the language of a graph analysis tool to define the metrics adds the advantage that tool support for the automatic calculation of metric values is immediately available. Graph query languages that have been used to specify design metrics include GraphLog [18] and GReQL [26].

As an example for the calculation of nontrivial metrics with RML, Fig. 9 shows a program that calculates Martin's instability metric for packages [27, chapter 20]. The metric is defined as $c_e/(c_a + c_e)$, where c_a is the number of classes outside the package that use classes inside the package, and c_e is the number of classes inside the package that use classes outside the package. In the RML program in Fig. 9, the `FOR` loop calculates c_a, c_e, and the value of the instability metric for each package.

3.5 Abstraction of Design Models

The examination of large systems requires views on different levels of abstraction. For example, class-level call graphs of large object-oriented programs have too many nodes and arcs to be understandable. So, views at higher levels of abstraction, e.g., package level, and means to zoom into parts of interest are needed. Usually, only low-level relationships, like method-level calls, are directly extracted

```
DegInh(super, sub, degsub) :=
                    Inherit(degsub, sub)
                  & Inherit(degsub, super)
                  & TC(Inherit(sub, super));
PRINT DegInh(super, sub, degsub);
```

Fig. 7. RML program for the detection of degenerate inheritance.

```
Know(super, sub) :=
       TC(Call(super, sub) | Contain(super, sub))
     & TC(Inherit(sub, super));
PRINT Know(super, sub);
```

Fig. 8. RML program for the detection of subclass knowledge.

```
Use(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
FOR p IN Package(x) {
  CaClass(x) := ! PackageOf(p,x)
                & EX(y, Use(x,y) & PackageOf(p,y));
  Ca := #(CaClass(x));
  CeClass(x) :=   PackageOf(p,x)
                & EX(y, Use(x,y) & !PackageOf(p,y));
  Ce := #(CeClass(x));
  PRINT p, " ", Ce / (Ca+Ce), ENDL;
}
```

Fig. 9. RML program for the calculation of the instability metric.

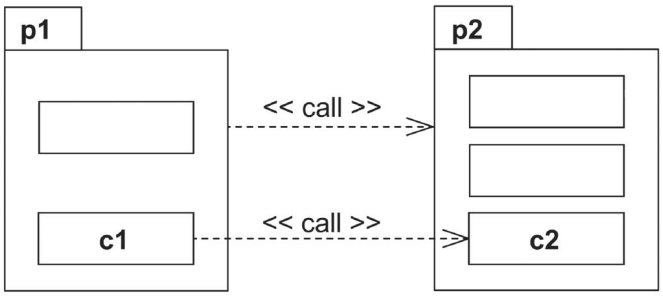

Fig. 10. A class-level call and the corresponding package-level call.

```
PCall(p1,p2) := EX(c1, EX(c2,   PackageOf(p1,c1)
                              & PackageOf(p2,c2)
                              & Call(c1,c2)));
PRINT PCall(p1,p2);
```

Fig. 11. RML program for lifting the `Call` relation to the package level.

from the source code, and the relationships between high-level entities have to be derived from these low-level relationships. This is called lifting and is a common application of relational calculators [1], [2].

Our example program derives the calls between packages from the calls between classes. Formally, a package `p1` calls a package `p2` if and only if there is a class `c1` in `p1` and a class `c2` in `p2` such that `c1` calls `c2` (see Fig. 10). This can be directly translated into the RML program in Fig. 11.

3.6 Other Applications

Querying and manipulating graphs and relations has many further applications in the analysis of software designs. The forward traversal of call and inheritance graphs is used to detect dead code, and the backward traversal is applied for change impact analysis [28], [2]. Computing and analyzing the difference between two graphs is useful for checking the conformance of the as-built architecture to the as-designed architecture [19], [2], [29], [22], [30] and for studying the evolution of software systems between different versions. Besides the detection of instances of design patterns and antipatterns in models of source code, graph pattern matching can also be used to extract scenarios [31], to identify code clones [32], to support the inductive inference of design patterns [33], [34], and to detect design problems in databases [35].

Up to now, we have mentioned only applications from software design. This is rather because we are most familiar with this area than because relational querying is useless elsewhere. For example, Berndl et al. [36] present an algorithm for computing the points-to relation between pointer variables and allocation sites (a problem from compiler optimization) which is based on the manipulation of relations, and an implementation that uses the same data structure (namely, BDDs) as our RML interpreter described in the next section.

4 Implementation

The main goals in the design of CrocoPat, our interpreter for RML programs, were efficiency and easy integration with other tools. Integration is facilitated by the import and export of relations in the simple Rigi Standard Format (RSF [37], [9]), which can be loaded into and saved from many reverse engineering tools and can be easily processed by scripts in common scripting languages. Efficient storage and manipulation of large relations is achieved by representing the relations as binary decision diagrams (BDDs [8]), a data structure that is widely used in computer-aided verification. The two sections below describe the representation of relations as BDDs and the manipulation of relations with BDDs.

4.1 Representation of Relations

An important problem in the representation of graphs and relations is efficiency. In Section 3, we used k-ary relations for detecting graph patterns with k nodes. Such relations can be very large: For a set M with $1,000$ elements, k-ary relations over M can have up to $1,000^k$ elements.

The data structure binary decision diagram (BDD [8]) exploits regularities in relations to represent them in a compressed form. It is important to note that not all relations are compressible and, thus, the use of BDDs does not improve the worst-case space complexity. However, experience in computer-aided verification shows that many practical relations are drastically compressible [38], and our experiments in Section 5 confirm this. In the following, we briefly introduce BDDs and give an example of how they represent large relations efficiently. For a more detailed introduction to BDDs, see [39].

A BDD is a rooted directed acyclic graph which is derived by reducing a binary decision tree. A binary decision tree has decision nodes, 0-terminal nodes, and 1-terminal nodes. Each decision node is labeled with a Boolean variable and has two children, called low child and high child. We only use *ordered* decision trees and *ordered* BDDs, which means that the Boolean variables occur in the same order on every path from the root to a terminal node.

A binary decision tree represents a relation over $\{0,1\}$, i.e., a set of bit vectors. The bit vectors represented by a decision tree correspond to the paths from the root node to the 1-terminals. The vector element that corresponds to a

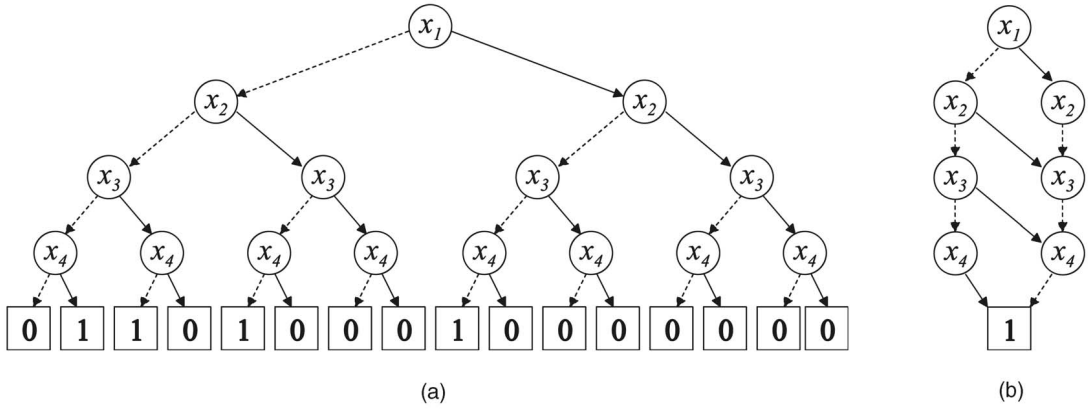

Fig. 12. Representations of the relation $Call$. (a) Decision tree representation. (b) BDD representation.

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.

The bottom-up application of the following two reduction rules transforms a binary decision tree into a BDD:

1. Merge any isomorphic subtrees.
2. Eliminate any node whose low child and high child are identical.

As an example, consider the following binary relation $Call$ over $\{A, B, C\}$:

$$Call = \{(A, B), (A, C), (B, A), (C, A)\}.$$

To represent this relation as binary decision tree or BDD, it must be transformed into a relation 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 decision tree representation of the relation $Call'$ (and, thus, of $Call$) is shown in Fig. 12a. Arcs to low children are represented as dashed lines, and arcs to high children are represented as solid lines. The four bit vectors in the relation $Call'$ correspond to the four paths from the root node to the four 1-terminal nodes in the decision tree. For example, the bit vector $(0, 0, 0, 1)$ corresponds to the path x_1, dashed line, x_2, dashed line, x_3, dashed line, x_4, solid line, 1-terminal.

Applying the two reduction rules to the decision tree results in the BDD in Fig. 12b. In this figure, arcs to the 0-terminal are omitted to avoid clutter. The reduction preserves the paths from the root node to the 1-terminal node and, thus, the represented relation.

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 $(c_1, c_2, \ldots, c_{k+1})$ with $(c_i, c_{i+1}) \in Call$ for all $i \in \{1, \ldots, k\}$. 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 eight 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 BDD for $k = 1$ in Fig. 12b with the BDD for $k = 3$ in Fig. 13a. The BDD for $k = 1$ can be transformed into the BDD for $k = 3$ by relabeling the x_3-nodes to x_7 and the x_4-nodes to x_8, and adding 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 BDD representation of a relation is only unique for a given variable order. In the BDD in Fig. 13a, the variable order (from the root to the terminals) is $x_1, x_2, \ldots, x_8$. The BDD in Fig. 13b represents the same relation with the variable order $x_1, x_3, x_5, x_7, x_2, x_4, x_6, x_8$. A comparison of the two BDDs shows that the BDD size is highly dependent on the variable order. In fact, it can even make the difference between linear and exponential BDD growth [39].

Because finding the optimal variable order of a BDD is algorithmically intractable [40], we have to apply heuristics. A particularly important heuristic is that strongly dependent variables should be placed closely in the variable order [41], [42], [43]. Thus, CrocoPat places all bits that encode the same tuple element at successive positions. The BDD in Fig. 13a conforms to this rule because x_2 is placed directly after x_1, x_4 directly after x_3, etc. The BDD in Fig. 13b violates the rule because there are several variables between x_1 and x_2, between x_3 and x_4, etc.

The detection of chains in call graphs is a simple example for the practically important problem of detecting graph patterns. It illustrates that the size of intermediate results obtained in searching graph patterns often grows exponentially with the size of the pattern and that BDDs can represent these large relations efficiently.

4.2 Manipulation of Relations

Besides memory, computation time is also a bottleneck in the manipulation of relations. Relational models of real systems can be large, and no polynomial-time algorithms are known for many practically important problems. For example, graph pattern detection—more precisely, the decision if there is a subgraph of one graph which is isomorphic to another graph—is NP-complete [44]. This does not preclude the existence of heuristics that are sufficiently efficient for many problem instances that occur

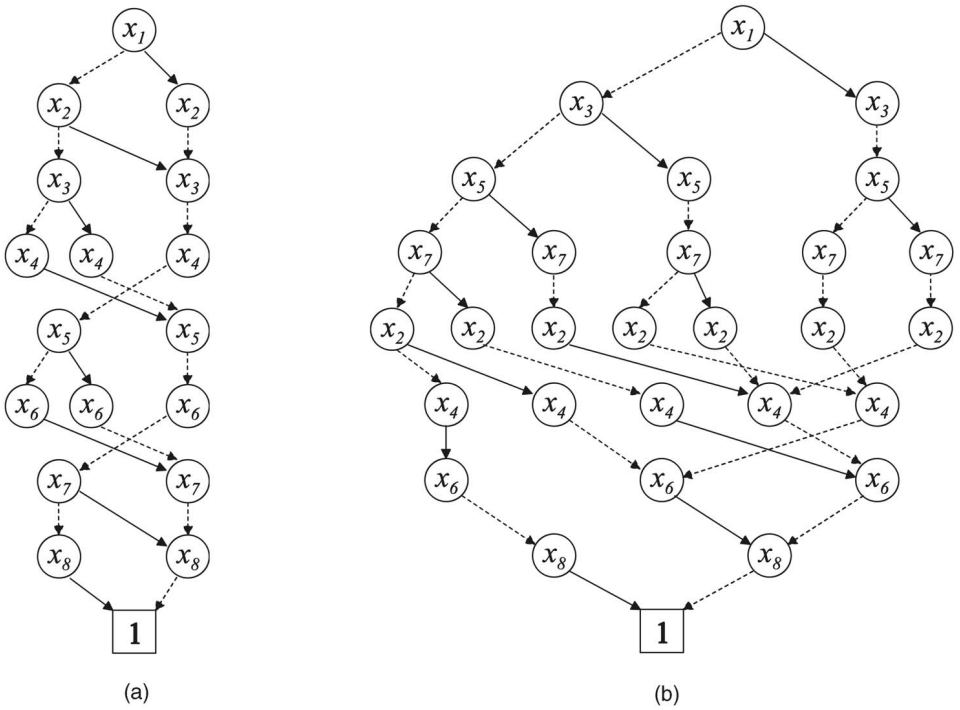

Fig. 13. BDD representations of all chains of three $Calls$. (a) Good variable order. (b) Bad variable order.

in practice. BDD-based algorithms are promising candidates for such heuristics.

The evaluation of RML's relational expressions (introduced in Section 2.1) in CrocoPat is organized into three layers:

1. The representation and manipulation of relations over $\{0,1\}$. This layer provides operations for the complement, the intersection, the union, and the comparison of relations, and for the quantification of bits.
2. The representation and manipulation of relations over strings. This layer provides operations for the quantification of entire attributes and for calculating the predefined relations for the lexicographical order.
3. The evaluation of RML's relational expressions. This layer provides operations for computing transitive closures and for the evaluation of compound expressions that contain more than one relational operator.

In the remainder of this subsection, we describe the three layers in turn.

The *bottom layer* is directly based on BDDs. Together with this data structure, Bryant introduced algorithms for elementary operations on relations over $\{0,1\}$ [8]. The worst-case time required for these algorithms is bounded by polynomials of the sizes of the operand BDDs. So, when the BDDs are small, their manipulation is efficient, even if they represent huge relations.

Reusable libraries for set manipulation that are based on BDDs and Bryant's algorithms are called BDD packages. The BDD package of CrocoPat was developed by the authors, but many other BDD packages are publicly available (see [45] for an overview). The implementation of an efficient BDD package requires several advanced techniques like hashing, caching, and garbage collection. A detailed description is beyond the scope of this article; we refer the reader to [46].

A benefit of the caching in the BDD package is that it frees the RML programmer from the responsibility for some optimizations and, thus, enables simpler and more explicit RML programs. For example, it is not necessary to factor out common subexpressions or constant expressions in a loop to avoid their repeated computation. Such repeated computations are avoided automatically by the BDD package, which looks up results in its cache instead of computing them twice.

The *second layer* employs the bit-level operations of the bottom layer to provide attribute-level operations. This seems fairly straightforward, but several implementation details are crucial for the overall efficiency. As an example, consider the computation of the BDD for the lexicographical comparison relation < over all strings. This relation has $n(n-1)/2$ elements, where n is the number of strings, so naive algorithms for computing its BDD use at least quadratic time. However, it turns out that the size of the BDD representation of the relation is only linear, thus it can be created in linear time.

The most interesting part of the *top layer* is the implementation of the transitive closure operator. In our experiments with different algorithms, we observed that the empirical complexity for practical graphs sometimes deviated strongly from the theoretical worst-case complexity, thus some algorithms with a relatively bad worst-case complexity were very competitive in practice.

```
Result(x,y) := R(x,y);
PrevResult(x,y) := FALSE(x,y);
WHILE (PrevResult(x,y) != Result(x,y)) {
  PrevResult(x,y) := Result(x,y);
  Result(x,z) := PrevResult(x,z)
    | EX(y, PrevResult(x,y) & PrevResult(y,z));
}
```

(a)

```
Result(x,y) := R(x,y);
Node(x) := EX(y, R(x,y)) & EX(y, R(y,x));
FOR node IN Node(x) {
  Result(x,y) := Result(x,y)
    | (Result(x,node) & Result(node,y));
}
```

(b)

```
Result(x,y) := R(x,y);
InvResult(x,y) := R(y,x);
Node(x) := EX(y, R(x,y)) & EX(y, R(y,x));
FOR node IN Node(x) {
  Result(x,y) := Result(x,y)
    | (InvResult(node,x) & Result(node,y));
  InvResult(x,y) := InvResult(x,y)
    | (Result(node,x) & InvResult(node,y));
}
```

(c)

Fig. 14. Algorithms for the transitive closure of a binary relation R. (a) Empirically fast algorithm. (b) Warshall's algorithm. (c) CrocoPat's algorithm, based on Warshall's algorithm.

Fig. 14a shows in RML notation a simple and empirically fast algorithm which performs a fixed point iteration to compute the transitive closure. After the ith iteration, the relation `Result` contains all pairs of nodes that are connected by a path of at most 2^i arcs.

The RML program in Fig. 14b is a straightforward implementation of the well-known transitive closure algorithm of Warshall [47]. It iterates through all nodes `node` of the graph that have ingoing and outgoing arcs. After each iteration, the relation `Result` contains all pairs of nodes that are connected by a path whose interior nodes were already considered in an iteration. This implementation needs less memory than the first because it uses no ternary relations, but much more time. The BDD representation works like a map from the start nodes of the arcs to their sets of adjacent end nodes, so finding all outgoing arcs of a node is fast, but finding all ingoing arcs—as in `Result(x,node)`—is slow.

The algorithm in Fig. 14c replaces this expensive expression with the more efficient `InvResult(node,x)`, where `InvResult` is the inverse of `Result`. In our experiments, this algorithm is about as fast as the first algorithm, but uses less memory, and is therefore the standard implementation of the transitive closure operator in CrocoPat.

5 Performance

To evaluate the efficiency of our tool implementation CrocoPat, we determined its runtime for two operations on graphs of different sizes. We chose operations that repeatedly occur in the analyses in Section 3 and that together account for a large part of the overall runtime of these analyses: the transitive closure and the detection of graph patterns. Through the use of these general building blocks of software analyses instead of concrete analysis scenarios, we expect to obtain more generalizable results.

Statements about the theoretical worst-case complexity of CrocoPat's algorithms for both problems are not difficult to derive, but such statements are not necessarily relevant in practice. Indeed, the worst-case runtime even of the best known algorithms is prohibitive for large graphs, but our goal was efficiency for graphs that occur in practical applications. For this reason, we evaluate the performance experimentally and use the call relations between classes of real software systems in our experiments. To illustrate the scaling behavior, we chose five systems of different sizes: JHotDraw 5.2, the AWT of the Java 2 Platform Standard Edition 1.4.2 (JDK 1.4.2 AWT), JWAM 1.8, the complete Java 2 Platform Standard Edition 1.4.2 (JDK 1.4.2), and Eclipse 2.1.2. Table 1 shows their characteristics. The call graphs were extracted from the

TABLE 1
Example Systems for Performance Evaluation

System	Classes	Calls	LOC
JHotDraw 5.2	171	655	17 819
JDK 1.4.2 AWT	354	981	142 838
JWAM 1.8	931	4 149	201 134
JDK 1.4.2	2 261	12 946	784 244
Eclipse 2.1.2	7 081	59 344	1 440 346

The column LOC gives the total number of carriage returns in the source code.

bytecode of the systems using the tool Sotograph.[1] For maximum comparability, the input data of the different tools is not only equivalent with respect to graph isomorphism, but also with respect to the node names and to the order of the arcs.

For each operation and each graph, we report the computation times of CrocoPat 2.1 and four other tools: the calculators for binary relations RelView 7.0.2 [3] and Grok 83 [1], the Prolog system Quintus Prolog 3.5 [6], and the relational database management system MySQL 4.0.15 [4]. We chose these tools because they were often applied for analyses similar to those in Section 3, and efficiency was a major concern in their development [3], [1, Section 5], [6, chapter 2.5], [4, chapter 1.2]. An additional advantage of Quintus Prolog in comparison with other Prolog implementations is its efficient implementation of a transitive closure predicate. Unlike MySQL, some commercial database management systems support recursive queries [48, Clause 7.12] that enable simpler (but not necessarily more efficient) queries for transitive closures, but these systems are not comparable to the four other tools in terms of complexity and expense.

The computation times are given in seconds on a Linux PC with 1.0 GHz AMD Athlon processor and 1,280 MB memory. Computations that took more than 5 hours were aborted. The loading of the input relation is *not* included in the given times to avoid biases caused by different input formats. To improve the performance of MySQL we used appropriate indexes and memory tables (i.e., hash tables stored in main memory, not on hard disc).

5.1 Transitive Closure

The programs for the computation of the transitive closure are shown in Fig. 15. The programs for CrocoPat, RelView, and Grok simply apply the respective transitive closure operators. Quintus Prolog provides an implementation of Warshall's transitive closure algorithm as predicate `warshall`. The Prolog program for computing the transitive closure includes some additional rather technical code for loading modules and building data structures (that is why it is not shown in Fig. 15), but the predicate `warshall` accounts for most of the runtime. For more information on transitive closure implementations in Prolog, see [49, chapter 5.4].

Queries for transitive closures cannot be expressed in basic SQL (i.e., relational algebra), as shown by Aho and Ullman [50]. So, we translated the body of the `WHILE` loop in Fig. 14a into the SQL script in Fig. 15d. This script was executed repeatedly until the fixed point was reached. Initially, the table `TCall` contains the relation `Call` and the table `TCallNew` is empty. After the ith iteration, the table `TCall` contains all pairs of classes that are connected by a sequence of at most 2^i calls.

1. http://www.software-tomography.com/.

```
TCall(x,y) := TC(Call(x,y));
```

(a)

```
TCall = trans(Call)
```

(b)

```
getdb $1
TCall := Call+
quit
```

(c)

```
INSERT INTO TCallNew (x,y)
  SELECT * FROM TCall;
INSERT INTO TCallNew (x,y)
  SELECT DISTINCT l.x, r.y
  FROM TCall l, TCall r
  WHERE l.y = r.x;
DELETE FROM TCall;

INSERT INTO TCall (x,y)
  SELECT DISTINCT * FROM TCallNew;
DELETE FROM TCallNew;
```

(d)

Fig. 15. Programs for the transitive closure of the relation `Call`. (a) CrocoPat. (b) RelView. (c) Grok. (d) MySQL.

The times in Table 2 show that CrocoPat, RelView, and Grok compute transitive closures much faster than Prolog and MySQL. For the largest graph (Eclipse 2.1.2), the BDD-based tools use significantly less memory than Grok, which is based on conventional data structures (CrocoPat 50 MB, RelView 100 MB, Grok 500 MB).

5.2 Graph Patterns

For the performance evaluation of graph pattern detection, we used cycles of different lengths as particularly simple, scalable, and practically relevant patterns. The detection of cycles of lengths greater than 2 cannot be naturally expressed with the languages of RelView and Grok because they are restricted to binary relations. The CrocoPat, Prolog, and MySQL queries for all cycles of length 4 are given in Fig. 16. Table 3 shows that, in our experiments, CrocoPat scaled better to large graphs and large patterns than Prolog, and Prolog scaled better than MySQL.

TABLE 2
Computation Times for the Transitive Closure of `Call`

System	CrocoPat	RelView	Grok	Prolog	MySQL
JHotDraw 5.2	0.05	0.06	0.02	0.06	0.38
JDK 1.4.2 AWT	0.11	0.17	0.14	0.82	11.1
JWAM 1.8	0.48	1.28	0.13	2.10	2.13
JDK 1.4.2	3.85	9.46	14.9	158	991
Eclipse 2.1.2	52.5	142	167	4 360	> 5 h

```
Cycle4(w,x,y,z) :=   Call(w,x) & Call(x,y)
                   & Call(y,z) & Call(z,w);
```

(a)

```
cycle4(W,X,Y,Z) :- cal(W,X), cal(X,Y),
                   cal(Y,Z), cal(Z,W).
```

(b)

```
INSERT INTO Cycle4
  SELECT DISTINCT C1.x, C2.x, C3.x, C4.x
  FROM Call C1, Call C2, Call C3, Call C4
  WHERE C1.y=C2.x AND C2.y=C3.x
    AND C3.y=C4.x AND C4.y=C1.x;
```

(c)

Fig. 16. Programs for cycles of length 4 of the relation `Call`. (a) CrocoPat. (b) Prolog. (c) MySQL.

TABLE 3
Computation Times for Cycles in `Call`

Tool	CrocoPat			Prolog			MySQL		
Cycle Length	4	6	8	4	6	8	4	6	8
JHotDraw 5.2	0.02	0.03	0.04	0.01	0.01	0.01	0.12	0.12	0.12
JDK 1.4.2 AWT	0.06	0.22	0.64	0.02	0.47	17.3	0.28	6.07	436
JWAM 1.8	0.16	0.26	0.36	0.02	0.12	0.72	0.20	1.31	7.04
JDK 1.4.2	1.09	5.72	19.1	0.96	85.4	7 430	11.4	1 070	> 5 h
Eclipse 2.1.2	5.01	22.7	116	2.85	123	8 190	24.3	1 850	> 5 h

6 Related Work

This section discusses languages and tools for querying and manipulating graphs and relations. We focus on approaches that have been used in software analysis and on their suitability for applications that we have identified as practically important but difficult, like graph pattern matching and the computation of transitive closures.

SQL is a well-known language for querying and manipulating relations. The lack of a transitive closure operator and the insufficient performance of relational database management systems for large graphs were already discussed in Section 5.

The logic programming language Prolog [7] has been used to detect design patterns and design problems in the tools Pat [12], Pattern-Lint [19], and Goose [21]. Its syntax for manipulating relations is similar to that of RML. However, RML is simpler than Prolog, and CrocoPat scales better to large relations than Prolog systems for the queries in Section 5.

Calculators for binary relational algebra that have been used to analyze software systems include Grok [1], RPA [2], and RelView [3]. As discussed in the introduction, the detection of graph patterns with more than two nodes cannot be naturally expressed in binary relational algebra. Grok was extended to provide graph pattern matching [31], but this resulted in a more complex language that still does not support other operations on n-ary relations.

The program understanding toolset GUPRO [51] provides the textual graph querying language GReQL [26]. GReQL focuses on querying binary relations (query *results* can have arbitrary arity), while RML also enables the creation and manipulation of arbitrary relations. Visual graph querying languages include GraphLog in the tool Hy+ [18], annotated graphs in IAPR [20], and a subset of UML in FUJABA [16].

The graph rewriting rule-based specification and rapid prototyping language PROGRES [52] has a purely textual and a combination of visual and textual notation. It is expressive, but also much more complicated than CrocoPat.

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., [38]). However, the only available BDD-based calculator for relations was RelView [3], which is limited to binary relations. The experimental results in Section 5 confirm the excellent performance of our BDD-based implementation in analyses of large software systems.

7 Conclusion

Modeling the structure of software systems as a graph enables the application of graph analysis tools for software analyses. However, existing tools for querying and manipulating graphs and relations do not fulfill all requirements of this application domain. In particular, calculators for binary relations are not powerful enough to detect general graph patterns with more than two nodes, and Prolog interpreters and relational database management systems are inefficient for practically important operations like the computation of transitive closures.

We proposed using the predicate calculus as basis of a language for the manipulation of n-ary relations and the data structure BDD for the efficient internal representation of n-ary relations. We evaluated our language RML by giving example programs for several analyses of object-oriented designs, and our tool implementation CrocoPat by performance measurements for typical expensive operations on large real-world software models. The evaluation confirmed that RML is sufficiently expressive and reasonably easy to use, and that CrocoPat scales well to the analysis of large software systems.

The tool CrocoPat is released as open source under GNU LGPL and is publicly available from the following website: http://www.software-systemtechnik.de/CrocoPat/.

References

[1] R.C. Holt, "Structural Manipulations Of Software Architecture Using Tarski Relational Algebra," *Proc. Fifth Working Conf. Reverse Eng. (WCRE 1998),* pp. 210-219, 1998.

[2] L.M.G. Feijs, R.L. Krikhaar, and R.C. van Ommering, "A Relational Approach to Support Software Architecture Analysis," *Software: Practice and Experience,* vol. 28, no. 4, pp. 371-400, 1998.

[3] R. Berghammer, B. Leoniuk, and U. Milanese, "Implementation of Relational Algebra Using Binary Decision Diagrams," *Proc. Sixth Int'l Conf. Relational Methods in Computer Science (RelMiCS 2001),* pp. 241-257, 2002.

[4] M. Widenius, D. Axmark, and MySQL AB, *MySQL Reference Manual.* Sebastopol, Calif.: O'Reilly, 2002.

[5] C.J. Date, *An Introduction to Database Systems,* eighth ed. Addison-Wesley, 2003.

[6] Swedish Inst. of Computer Science, *Quintus Prolog User's Manual.* 2003.

[7] W.F. Clocksin and C.S. Mellish, *Programming in Prolog,* fifth ed. Springer-Verlag, 2003.

[8] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677-691, Aug. 1986.

[9] D. Beyer and A. Noack, "Crocopat 2.1 Introduction and Reference Manual," Technical Report UCB//CSD-04-1338, Computer Science Division (EECS), Univ. of California, Berkeley, 2004, http://arxiv.org/abs/cs/0409009.

[10] H.-D. Ebbinghaus, J. Flum, and W. Thomas, *Mathematical Logic,* second ed. Springer-Verlag, 1994.

[11] H. Fahmy, R.C. Holt, and J.R. Cordy, "Wins and Losses of Algebraic Transformations of Software Architectures," *Proc. 16th Int'l Conf. Automated Software Eng. (ASE 2001),* pp. 51-60, 2001.

[12] C. Krämer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software," *Proc. Third Working Conf. Reverse Eng. (WCRE 1996),* pp. 208-215, 1996.

[13] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design Pattern Recovery in Object-Oriented Software," *Proc. Sixth Int'l Workshop Program Comprehension (IWPC 1998),* pp. 153-160, 1998.

[14] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," *Proc. 11th Int'l Workshop Program Comprehension (IWPC 2003),* pp. 94-103, 2003.

[15] R.K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-Based Reverse-Engineering of Design Components," *Proc. 21st Int'l Conf. Software Eng. (ICSE 1999),* pp. 226-235, 1999.

[16] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh, "Towards Pattern-Based Design Recovery," *Proc. 24th Int'l Conf. Software Eng. (ICSE 2002),* pp. 338-348, 2002.

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

[18] A.O. Mendelzon and J. Sametinger, "Reverse Engineering by Visualizing and Querying," *Software—Concepts and Tools,* vol. 16, no. 4, pp. 170-182, 1995.

[19] M. Sefika, A. Sane, and R.H. Campbell, "Monitoring Compliance of a Software System with Its High-Level Design Models," *Proc. 18th Int'l Conf. Software Eng. (ICSE 1996),* pp. 387-396, 1996.

[20] R. Kazman and M. Burth, "Assessing Architectural Complexity," *Proc. Second Euromicro Conf. Software Maintenance and Reeng. (CSMR 1998),* pp. 104-112, 1998.

[21] O. Ciupke, "Automatic Detection of Design Problems in Object-Oriented Reengineering," *Proc. 30th Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS 1999),* pp. 18-32, 1999.

[22] H. Fahmy and R.C. Holt, "Software Architecture Transformations," *Proc. Int'l Conf. Software Maintenance (ICSM 2000),* pp. 88-96, 2000.

[23] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1996.

[24] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed. Boston, Mass.: PWS, 1997.

[25] T. Mens and M. Lanza, "A Graph-Based Metamodel for Object-Oriented Software Metrics," *Electronic Notes in Theoretical Computer Science,* vol. 72, no. 2, 2002.

[26] B. Kullbach and A. Winter, "Querying as an Enabling Technology in Software Reengineering," *Proc. Third European Conf. Software Maintenance and Reeng. (CSMR 1999),* pp. 42-50, 1999.

[27] R.C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.

[28] Y.-F. Chen, E.R. Gansner, and E. Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection," *IEEE Trans. Software Eng.,* vol. 24, no. 9, pp. 682-694, 1998.

[29] K. Mens, R. Wuyts, and T. D'Hondt, "Declaratively Codifying Software Architectures Using Virtual Software Classifications," *Proc. 29th Int'l Conf. Technology of Object-Oriented Languages and Systems—Europe (TOOLS 1999),* pp. 33-45, 1999.

[30] G.C. Murphy, D. Notkin, and K.J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Trans. Software Eng.,* vol. 27, no. 4, pp. 364-380, Apr. 2001.

[31] J. Wu, A.E. Hassan, and R.C. Holt, "Using Graph Patterns to Extract Scenarios," *Proc. 10th Int'l Workshop Program Comprehension (IWPC 2002),* pp. 239-247, 2002.

[32] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth Working Conf. Reverse Eng. (WCRE 2001),* pp. 301-309, 2001.

[33] 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 Dept., Univ. of Maryland, 1996.

[34] P. Tonella and G. Antoniol, "Object Oriented Design Pattern Inference," *Proc. Int'l Conf. Software Maintenance (ICSM 1999),* pp. 230-238, 1999.

[35] M. Blaha, "A Copper Bullet for Software Quality Improvement," *Computer,* vol. 37, no. 2, pp. 21-25, Feb. 2004.

[36] M. Berndl, O. Lhoták, F. Qian, L.J. Hendren, and N. Umanee, "Points-To Analysis Using BDDs," *Proc. Conf. Programming Language Design and Implementation (PLDI 2003),* pp. 103-114, 2003.

[37] K. Wong, *Rigi User's Manual, Version 5.4.4,* 1998, http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/.
[38] 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,* vol. 98, no. 2, pp. 142-170, 1992.
[39] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys,* vol. 24, no. 3, pp. 293-318, 1992.
[40] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs Is NP-Complete," *IEEE Trans. Computers,* vol. 45, no. 9, pp. 993-1002, Sept. 1996.
[41] C.L. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams," *IEEE Trans. Computer-Aided Design,* vol. 10, no. 8, pp. 1059-1066, 1991.
[42] S.-W. Jeong, B. Plessier, G.D. Hachtel, and F. Somenzi, "Variable Ordering and Selection for FSM Traversal," *Proc. Int'l Conf. Computer-Aided Design (ICCAD 1991),* pp. 476-479, 1991.
[43] A. Aziz, S. Tasiran, and R.K. Brayton, "BDD Variable Ordering for Interacting Finite State Machines," *Proc. 31st Design Automation Conf. (DAC 1994),* pp. 283-288, 1994.
[44] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman, 1979.
[45] G. Janssen, "A Consumer Report on BDD Packages," *Proc. 16th Symp. Integrated Circuits and Systems Design (SBCCI 2003),* pp. 217-222, 2003.
[46] K.S. Brace, R.L. Rudell, and R.E. Bryant, "Efficient Implementation of a BDD Package," *Proc. 27th Design Automation Conf. (DAC 1990),* pp. 40-45, 1990.
[47] S. Warshall, "A Theorem on Boolean Matrices," *J. ACM,* vol. 9, no. 1, pp. 11-12, 1962.
[48] ANSI/ISO/IEC 9075:1999, "Information Technology—Database Languages—SQL—Part 2: Foundation," 1999.
[49] R.A. O'Keefe, *The Craft of Prolog.* MIT Press, 1990.
[50] A.V. Aho and J.D. Ullman, "Universality of Data Retrieval Languages," *Proc. Sixth Ann. ACM Symp. Principles of Programming Languages (POPL 1979),* pp. 110-120, 1979.
[51] J. Ebert, B. Kullbach, V. Riediger, and A. Winter, "GUPRO—Generic Understanding of Programs," *Electronic Notes in Theoretical Computer Science,* vol. 72, no. 2, 2002.
[52] D. Blostein and A. Schürr, "Computing with Graphs and Graph Transformations," *Software: Practice and Experience,* vol. 29, no. 3, pp. 197-217, 1999.

Dirk Beyer received the Diploma degree (1998) and Doctoral degree (2002) in computer science from the Brandenburg University of Technology at Cottbus, Germany. He is a postdoctoral researcher in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. In 1998, he was a software engineer with Siemens AG at Dresden. His research interests include structural analysis and comprehension of large software systems, software model checking, and formal verification of real-time systems. He is a member of the IEEE and the IEEE Computer Society.

Andreas Noack received the Diploma degree in computer science from the Brandenburg University of Technology at Cottbus, Germany (2000). He is a research and teaching assistant in the Department of Computer Science at the Brandenburg University of Technology. His research focuses on the analysis and visualization of large object-oriented software systems. He is a member of the IEEE Computer Society.

Claus Lewerentz received the Diploma degree in computer science from Munich Technical University (1983) and the Doctoral degree from RWTH Aachen (1988). He is a full professor of software engineering at the Brandenburg University of Technology at Cottbus, Germany. He is head of the Software and Systems Engineering Research Group. Jointly, he is manager of the Fraunhofer FIRST Software Quality Lab and a cofounder of Software Tomography Inc. His research interests are in the areas of the construction, analysis, and visualization of large software systems and software quality. He is a member of the German Informatics Society (GI), the German Association of Engineers (VDI), and IFIP.