Relational Programming with CrocoPat

Dirk Beyer
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

ABSTRACT

Many structural analyses of software systems are naturally formalized as relational queries, for example, the detection of design patterns, patterns of problematic design, code clones, dead code, and differences between the as-built and the as-designed architecture. This paper describes CrocoPat, an application-independent tool for relational programming. Through its efficiency and its expressive language, CrocoPat enables practically important analyses of real-world software systems that are not possible with other graph analysis tools, in particular analyses that involve transitive closures and the detection of patterns in graphs. The language is easy to use, because it is based on the well-known first-order predicate logic. The tool is easy to integrate into other software systems, because it is a small command-line tool that uses a simple text format for input and output of relations.

Categories and Subject Descriptors: D.1.6 [Programming Techniques]: Logic Programming; G.2.2.a [Discrete Mathematics]: Graph Theory—*Graph algorithms*; E.1.d [Data Structures]: Graphs and networks; D.2.7.m [Software Engineering]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.11 [Software Architectures]: Data abstraction; D.2.13 [Reusable Software]: Reusable libraries; G.4 [Mathematical Software]: Algorithm design; Efficiency;

General Terms: Algorithms, Design, Languages

Keywords: Software analysis, Predicate logic, Relational algebra, BDD, Graph models, Pattern matching, Transitive closure

1. INTRODUCTION

Graphs and other relations are often used to define abstract models of software systems (e.g., control-flow, dependency, call, and inheritance graphs), and many structural software analyses involve complex computations on these relational models (e.g., impact analysis, design pattern matching, detection of cyclic dependencies). Particularly complex operations on software graphs are pattern matching and transitive closure computation.

CrocoPat[1] is a tool for efficient relational calculation. Its relational manipulation language (RML) is a simple but expressive language. It provides the usual control flow structures of an imperative programming language, and the expressions over relations are those of predicate logic. The tool implementation represents relations as binary decision diagrams (BDD) [9], which are well-known as a data structure for compact representation of large relations [10].

Relational Programming. The language RML is based on predicate logic and is therefore similar to Prolog. However, RML does not follow the paradigm of logic programming, it is rather an imperative programming language. Instead of being declarative and inference-based, it is operational and executes the program statement by statement. This is why we prefer the term *relational programming*.

The idea of an imperative relational programming language is old [26], but such a language could not be used in practice so far because the approach was lacking an efficient interpreter (cf. [34] for a discussion on SETL). Although many software analysis tools are based on relational methods, there was no effort so far to provide a general purpose tool and application-independent language. CrocoPat is such an interpreter for a general purpose relational programming language, which is efficient especially for the kind of computations that are needed in software analyses.

Relational programming has many applications in *software analysis*, (e.g., relation lifting, checking of design rules, cycle analysis, impact analysis, detecting unused components and dead code, detecting similar code and similar classes, points-to analysis), in *theory* (e.g., minimization of deterministic automata, Gaussian elimination), and for *graph algorithms* in general (e.g., reachability analysis, graph pattern matching, transitive closure, shortest paths, SCC analysis). Some of the more software-analysis related applications were thoroughly examined in our case studies to show the usability (in terms of expressiveness, performance, and easiness of use) of CrocoPat [8].

Example. Let us examine a small pattern-matching example. Consider a software system that is modeled by its inheritance and containment graphs, and the task is to analyze how many instances of the design pattern Composite are used in the design of the system. Figure 1 shows the class diagram of the Composite design pattern. To identify all possible instances of the pattern, we have to compute all triples of three classes (Component, Composite, Leaf), such that (1) Composite and Leaf are subclasses of Component, (2) Composite contains Component, and (3) Leaf does not contain Component.

If the inheritance and containment graphs are represented by the two binary predicates *Inherit* and *Contain*, we can

[1] Project web site: `http://mtc.epfl.ch/~beyer/CrocoPat`
Download: `http://directory.fsf.org/CrocoPat.html`

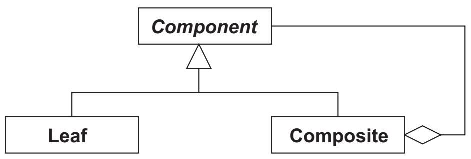

Figure 1: Composite design pattern, simplified

naturally encode the problem as a first-order expression with three free attributes component, composite, and leaf: $\mathit{Inherit}(\mathsf{composite},\mathsf{component}) \wedge \mathit{Inherit}(\mathsf{leaf},\mathsf{component}) \wedge \mathit{Contain}(\mathsf{composite},\mathsf{component}) \wedge \neg \mathit{Contain}(\mathsf{leaf},\mathsf{component})$. We can translate this expression one-to-one into an RML expression (cf. Section 2). Then we can feed CrocoPat with the two input graphs and an RML program that prints the result of the above expression, to output all instances of the pattern in the software model.

Related Work. There are three alternatives to relational programming: relational algebra with SQL as query language and database management systems as interpreter (cf. [1, 13, 39]), first-order predicate logic with Prolog as programming language and Prolog systems as interpreter (cf. [31, 37]), and graph manipulation, supported by several tools, each with its own language (most prominently Grok [18], RPA [15], and RelView [4]). The latter approaches are restricted to graphs (binary relations), and cannot be applied to relations of arbitrary arity in a natural way. SQL servers and Prolog interpreters are not efficient enough for the kind of relational computations that occur in software analyses[2]. (They are designed for different objectives for which they *are* efficient, e.g., for data retrieval or logical deduction, respectively.) Aho and Ullman argued that an operator for transitive closure is needed for many database applications, which is not supported in relational algebra and calculus [1]. An imperative relational programming language for the purpose of writing abstract algorithms based on relations was proposed by MacLennan [26] (also cf. SETL [34]). CrocoPat provides a language that is as powerful as the previous approaches, but adds a convenient operator for transitive closure; and the interpreter is efficient for general purpose relational computation, because it is based on BDD technology.

2. CROCOPAT — OVERVIEW

CrocoPat is a command-line tool, designed as a filter (in the sense of the pipes-and-filters architecture, which is typical for Unix tools). It takes as input a program file and a stream of input relations, and outputs user messages and a stream of output relations. In the following we highlight some of the most important features of the tool and its language (cf. [7, 8] for more details).

Language. A simple relational program is a sequence of statements, where each statement is either an assignment or a print statement. There are two types of variables: string and relation variables.[3] The relation variables range over sets of tuples of strings, and have an arbitrary arity (relations of arity 0 are the Boolean values true and false, unary relations are sets, and binary relations are graphs). There is no need to declare the type of a variable; it is determined by the value of the first assignment to the variable. For the convenient and structured expression of more complex programs, RML also has constructs (`IF`, `WHILE`, `FOR`) for the conditional execution and iteration of statements.

```
Statement:  RelVar '(' StrExp ',' StrExp ')'
            ':=' RelExp ';'
          | 'PRINT' RelExp ';'

RelExp:     RelVar '(' StrExp ',' StrExp ')'
          | 'TRUE' '(' StrExp ')'
          | 'FALSE' '(' StrExp ')'
          | RelExp '&' RelExp       // conjunction
          | RelExp '|' RelExp       // disjunction
          | '!' RelExp              // negation
          | 'EX' '(' Attr ',' RelExp ')'
          | 'FA' '(' Attr ',' RelExp ')'
          | '@'regex '(' StrExp ')' // matching

StrExp:     StrLit | Attr
```

Figure 2: Partial syntax of RML

The core construct of the language is the *relational expression*, which is similar to an expression in first-order predicate logic. However, RML provides in addition an operator for transitive closure, an operator for regular-expression matching, and operators for comparison of relations, but does not include functions.

In the right-hand side expression of an assignment, every identifier must either be a relation variable and have been previously assigned a relation, or it must be a string variable and have been previously assigned a string, or it must be an attribute that is quantified or occurs free. RML allows relations of arbitrary arity, but for simplicity, let us restrict this discussion to binary relation variables. Also, let us write x and y for the values of the attributes `x` and `y`, and R for the set of pairs denoted by the binary relation variable `R`. Then the expression `R(x,y)` evaluates to true iff $(x, y) \in R$. To assign a new value to the relation variable `R` we write "`R(x,y) := e`" short for "for all `x,y` let `R(x,y) := e`," where `e` is a relational expression that must contain free occurrences of `x` and `y`. The expression "`@regex(x)`" evaluates to true iff x matches the regular expression `regex` (i.e., `@regex` denotes the set of all strings that match the regular expression `regex`).

Each print statement has as argument a relational expression, with possibly some free occurrences of attributes. The result is a print-out of all value assignments to the free attributes that make the expression true. For example, "`PRINT R(x,y)`" outputs all pairs (x, y) of strings such that $(x, y) \in R$. The output of user messages is also possible.

The grammar for a simple subset of RML is shown in Figure 2. The nonterminals `Attr` and `RelVar` refer to any RML identifier; `StrLit` is a string literal; and `regex` is a Unix regular expression. A complete definition of the syntax and semantics of RML is given in the reference manual [7].

Input/Output. Besides the RML file containing the program, CrocoPat expects input relations from the standard input stream (if not switched off by a command-line option). To enable easy data exchange and integration of CrocoPat

[2]A performance comparison of CrocoPat with Quintus Prolog 3.5 [37] and MySQL 4.0.15 [39], but also with Grok 83 [18] and RelView 7.0.2 [4], shows that CrocoPat outperforms the others [8].

[3]RML supports numerical variables as well, for convenient calculation with numbers. But this is not of conceptual interest.

into other tools, the format for input relations is the Rigi Standard Format (RSF) [40]. RSF is a simple text format where each line represents one tuple of a relation (as whitespace-separated list of strings). The first string of a line is the relation variable that denotes the relation in the RML program; all other strings are elements of the tuple, and their number is the arity of the relation. One RSF stream can define an arbitrary number of relations.

The output of an RML program can be printed to the standard (or error) output stream, or written to a file. The user messages of an RML program are usually printed to standard output and output relations are redirected to files (determined by the programmer in the PRINT statement).

Examples. Let us consider the design pattern example from the introduction. The following RML program computes all triples for which the relational expression evaluates to true in the first statement. The second statement prints these triples to the standard output (i.e., on the screen).

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

The above RML program expects as input two relations Inherit and Contain. An example for such an RSF input (which contains pattern instance (C,A,B)) is the following:

```
Inherit A   C
Inherit B   C
Contain A   C
```

As a second example, let us consider the following RML program for computing the transitive closure[4] using an algorithm called iterative squaring [10]:

```
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));
}
```

The program expects a relation R as input. The program contains only relation variables (R, Result, and PrevResult) and attributes (x, y, z), no string variables. Result represents always the current intermediate result of the computation, and PrevResult the result of the previous iteration. (This is why Result is initialized with the given relation and PrevResult with the empty relation.) The iteration in the body of the WHILE loop is executed until the fixed point is reached, i.e., the result of the current result equals the previous result. The forward step in the second statement of the body can be read as (we write x, y, z for concrete values of the attributes x, y, z, again) "for all x and z, it holds that $(x, z) \in \mathit{Result}$ iff $(x, z) \in \mathit{PrevResult}$ or there exists a y such that $(x, y) \in \mathit{PrevResult}$ and $(y, z) \in \mathit{PrevResult}$".

[4]Although this is not necessary in practice because RML provides an operator for transitive closure (TC). CrocoPat's built-in operator uses a variant of Warshall's algorithm as standard implementation because it consumes much less memory, and therefore is more robust in extreme cases.

Architecture. CrocoPat's architecture consists of three layers. The bottom layer is an own implementation of a standard package for binary decision diagrams (BDDs), which implements operations for the intersection, union, complement, comparison, and quantification on the bit level. BDDs represent relations symbolically, i.e., the BDD representation of a large relation can be very small, and the complexity of the operations depends on the size of the operand BDDs, not on the size of the represented relations [9, 10].

The middle layer implements the abstraction from the bit level to the string level, e.g., the encoding of strings to assignments over Boolean variables, quantification of entire attributes, and the computation of the predefined relations for lexicographical ordering.

The top layer is the actual interpreter. It consists of a parser front-end, and an evaluation unit for relational expressions that translates the operation symbols in the syntax tree into executions of (compound) operations from the middle layer. Besides this, it implements the transitive closure operator.

Availability. The relational calculator CrocoPat is free software, released under the GNU LGPL license. The current stable version (release 2.1.3) is available online at http://directory.fsf.org/CrocoPat.html. The web site provides the tool (source code and binaries), example programs (RML files), and data (RSF files). The user's guide and reference manual is also provided on the web site. An API description is included directly in the header files, to enable reuse of the tool on library level and for extensions.

3. APPLICATIONS

Computing with relations is part of many software analysis tools. However, some of them are limited in their functionality and performance because the tools are lacking an efficient computation machinery for actually performing the tasks. We propose, for future development of such tools, to out-source the relational queries to a tool (or library) like CrocoPat, which can perform these queries more efficiently, due to its highly optimized engine. The tool developers of software reengineering and analysis tools can then focus more on implementing their core functionality, instead of struggling with the low-level details of improving performance for the relational calculations. In the following we list some applications that we are aware of. More references to the literature are given in the technical report [7].

Graph pattern detection is widely applied to detect good design (design patterns) and design problems (anti-patterns) of software systems. This was also the first application of CrocoPat [6], because existing tools were either limited to binary relations (e.g., Grok [18], RPA [15], and RelView [4]), or consumed too much time (relational database servers and Prolog interpreters). Many researchers applied relational methods to automatic detection of implementation patterns, object-oriented design patterns, architectural styles, potential design problems, code clones, inductive inference of design patterns [2, 12, 14–18, 21–24, 27, 30, 32–33, 35–36, 38].

Transitive closure computation is ingredient of some of the above analyses, but is also crucial for dead code and change impact analysis [11, 15]. Computing the difference between two graphs is necessary for checking the conformance of the as-built architecture to the as-designed architecture [14, 15, 28, 29, 35]. Another straightforward application of relational methods is the lifting and lowering of relations [14, 15] to get new abstraction levels, and the calculation of software metrics (e.g., [25, 27]).

Efficient calculators for relations are important for program analyses like points-to analysis [5], and for the implementation of general graph algorithms. Also software modeling languages like Alloy heavily rely on efficient relational computation [19, 20].

Since relational programming is an efficient way of prototyping relation-intensive algorithms, especially verification algorithms, CrocoPat was recently used to prototype algorithms for symbolic invariant verification [3].

4. SUMMARY

CrocoPat's programming language is a simple but general and expressive language for relational computations. It enables to write algorithms over relations of arbitrary arity in a concise way, and it is easy to understand and learn because it is based on first-order predicate logic. The tool CrocoPat is an efficient interpreter for that language, because it is based on a fine-tuned library for relational operations. It is easy to use and install, because it consists of only one executable file and provides only a few command-line options.

CrocoPat was developed in the hope that it improves the productivity of other software engineers, especially those whose task is to analyze software. Besides using CrocoPat directly as a tool for analyses, developers of reengineering and analysis tools can out-source subtasks that involve computation with relations to CrocoPat. They can reduce their burden of implementing the algorithms by formulating them as abstract relational programs.

Due to its simple user interface it is easy to integrate CrocoPat into other tools, and its library can serve — through its well-documented source code interface— as a high-level library for relational computation. This is a convenient alternative to using a standard BDD package directly, because the developer does not need to take care of the binary encoding himself.

Acknowledgement. We thank Andreas Noack for helpful discussions about the language and the tool design, and for his valuable collaboration on applying CrocoPat to software design analysis [8].

5. REFERENCES

[1] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. POPL*, pages 110–120. ACM, 1979.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proc. IWPC*, pages 153–160. IEEE, 1998.

[3] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proc. ICSE*. ACM Press, 2006.

[4] R. Berghammer, B. Leoniuk, and U. Milanese. Implementation of relational algebra using binary decision diagrams. In *Proc. RelMiCS'01*, LNCS 2561, pages 241–257. Springer, 2002.

[5] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proc. PLDI*, pages 103–114. ACM, 2003.

[6] D. Beyer and C. Lewerentz. CrocoPat: Efficient pattern analysis in object-oriented programs. In *Proc. IWPC*, pages 294–295. IEEE, 2003.

[7] D. Beyer and A. Noack. Crocopat 2.1 Introduction and reference manual. Technical Report CSD-04-1338, University of California, Berkeley, 2004.

[8] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Engineering*, 31(2):137–149, 2005.

[9] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[10] 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.

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

[12] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proc. TOOLS*, pages 18–32. IEEE, 1999.

[13] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[14] H. Fahmy and R. C. Holt. Software architecture transformations. In *Proc. ICSM*, pages 88–96. IEEE, 2000.

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

[16] M. T. Harandi and J. Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, 1990.

[17] J. Hartman. Understanding natural programs using proper decomposition. In *Proc. ICSE*, pages 62–73. IEEE, 1991.

[18] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proc. WCRE*, pages 210–219. IEEE, 1998.

[19] D. Jackson. Automating first-order relational logic. In *Proc. FSE*, pages 130–139. ACM, 2000.

[20] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[21] R. Kazman and M. Burth. Assessing architectural complexity. In *Proc. CSMR*, pages 104–112. IEEE, 1998.

[22] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Proc. ICSE*, pages 226–235. ACM, 1999.

[23] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. WCRE*, pages 208–215. IEEE, 1996.

[24] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. WCRE*, pages 301–309. IEEE, 2001.

[25] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proc. CSMR*, pages 42–50, 1999.

[26] B. J. MacLennan. Overview of relational programming. *SIGPLAN Notices*, 18(3):36–45, 1983.

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

[28] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proc. TOOLS*, pages 33–45. IEEE, 1999.

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

[30] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. ICSE*, pages 338–348. ACM, 2002.

[31] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, 1990.

[32] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.

[33] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[34] J. T. Schwartz. Automatic data structure choice in a language of very high level. *Commun. ACM*, 18(12):722–728, 1975.

[35] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proc. ICSE*, pages 387–396. IEEE, 1996.

[36] 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, University of Maryland, 1996.

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

[38] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proc. ICSM*, pages 230–238. IEEE, 1999.

[39] M. Widenius, D. Axmark, and MySQL AB. *MySQL Reference Manual*. O'Reilly, Sebastopol, CA, 2002.

[40] K. Wong. *Rigi User's Manual, Version 5.4.4*, 1998.