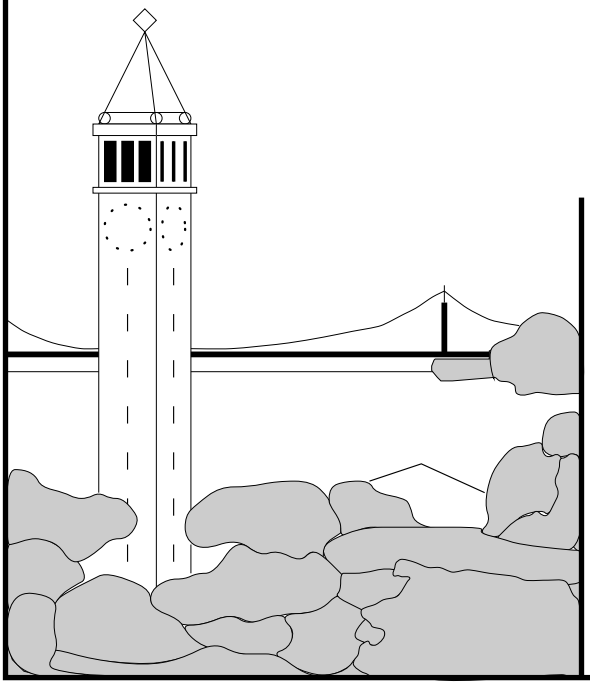
CrocoPat 2.1 Introduction and Reference Manual

Dirk Beyer *Andreas Noack*



Report No. UCB//CSD-04-1338

July 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

CrocoPat 2.1 Introduction and Reference Manual*

Dirk Beyer[1] and Andreas Noack[2]

[1] Electronics Research Laboratory
Department of Electrical Engineering and Computer Sciences
College of Engineering
University of California at Berkeley
Berkeley, CA 94720-1770, U.S.A.
beyer@eecs.berkeley.edu

[2] Software Systems Engineering Research Group
Department of Computer Science
Brandenburg University of Technology at Cottbus
PO Box 10 13 44, 03013 Cottbus, Germany
an@informatik.tu-cottbus.de

Abstract. CrocoPat is an efficient, powerful and easy-to-use tool for manipulating relations of arbitrary arity, including directed graphs. This manual provides an introduction to and a reference for CrocoPat and its programming language RML. It includes several application examples, in particular from the analysis of structural models of software systems.

1 Introduction

CrocoPat is a tool for manipulating relations, including directed graphs (binary relations). CrocoPat is powerful, because it manipulates relations of arbitrary arity; it is efficient in terms of time and memory, because it uses the data structure binary decision diagram (BDD, [Bry86,Bry92]) for the internal representation of relations; it is fairly easy to use, because its language is simple and based on the well-known predicate calculus; and it is easy to integrate with other tools, because it uses the simple and popular Rigi Standard Format (RSF) as input and output format for relations. CrocoPat is free software (released under LGPL) and can be obtained from `http://www.software-systemtechnik.de/CrocoPat`.

Overview. CrocoPat is a command line tool which interprets programs written in the Relation Manipulation Language (RML). Its inputs are an RML program and relations in the Rigi Standard Format (RSF), and its outputs are relations in RSF and other text produced by the RML program. The programming language RML, and the input and output of relations from and to RSF files are introduced with the help of many examples in Section 2. Section 3 describes advanced programming techniques, in particular for improving the performance of RML programs and for circumventing limitations of RML. The manual concludes with references of CrocoPat's command line options in Section 4, of RSF in Section 5, and of RML in Section 6. The RML reference includes a concise informal description of the semantics, and a formal description of the syntax and the core semantics.

Applications. CrocoPat was originally developed for analyzing graph models of software systems, and in particular for finding patterns in such graphs [BNL03]. Existing tools were not appropriate for this task, because they were limited to binary relations (e.g. Grok [Hol98], RPA [FKvO98], and RelView [BLM02]), or consumed too much time or memory (e.g. relational database management systems and Prolog interpreters). Applications of graph pattern detection include

- the detection of implementation patterns [RW90,HN90,Har91,Qui94], object-oriented design patterns (Section 2.4, [MS95,KP96,AFC98,KSRP99,NSW+02]), and architectural styles [Hol96],
- the detection of potential design problems (Section 2.4, [MS95,SSC96,FKvO98,KB98,Ciu99,FH00]),

* This research was supported in part by the NSF grants CCR-0234690 and ITR-0326577, and the DFG grant BE 1761/3-1.

– the inductive inference of design patterns [SMB96,TA99],
– the identification of code clones [Kri01],
– the extraction of scenarios from models of source code [WHH02], and
– the detection of design problems in databases [Bla04].

The computation of transitive closures of graphs – another particular strength of CrocoPat – is not only needed for the detection of some of the above patterns, but has also been applied for dead code detection and change impact analysis [CGK98,FKvO98]. Computing and analyzing the difference between two graphs supports checking the conformance of the as-built architecture to the as-designed architecture [SSC96,FKvO98,MWD99,FH00,MNS01], and studying the evolution of software systems between different versions. Calculators for relations have also been used to compute views of systems on different levels of abstraction by lifting and lowering relations [FKvO98,FH00], and to calculate software metrics (Section 2.5, [MS95,KW99]).

Although we are most familiar with potential applications in the analysis of software designs, we are confident that CrocoPat can be beneficial in many other areas. For example, calculators for relations have been used for program analyses like points-to analysis [BLQ+03], and for the implementation of graph algorithms (Section 2.3, [BBS97]).

2 RML Tutorial

This section introduces RML, the programming language of CrocoPat, on examples. The core of RML are relational expressions based on first-order predicate calculus, a well-known, reasonably simple, precise and powerful language. Relational expressions are explained in Subsection 2.1, and additional examples involving relations of arity greater than two are given in Subsection 2.4. Besides relational expressions, the language includes control structures, described in Subsection 2.3, and numerical expressions, described in Subsection 2.5. The input and output of relations is described in Subsection 2.2. A more concise and a more formal specification of the language can be found in Section 6.

Although the main purpose of this section is the introduction of the language, some of the application examples may be of interest by themselves. In Subsection 2.3, simple graph algorithms are implemented, and in the Subsections 2.4 and 2.5 the design of object-oriented software systems is analyzed.

2.1 Relational Expressions

This subsection introduces relational expressions using relationships between people as example. Remember that n-ary relations are sets of ordered n-tuples. In this subsection, we will only consider the cases $n = 1$ (unary relations) and $n = 2$ (binary relations, directed graphs). CrocoPat manipulates tuples of strings, thus unary relations in CrocoPat are sets of strings, and binary relations in CrocoPat are sets of ordered pairs of strings.

Adding Elements. The statement

```
Male("John");
```

expresses that `John` is male. (In some languages, e.g. the logic programming language Prolog [CM03], such statements are called facts.) It adds the string `John` to the unary relation `Male`. Because each relation variable initially contains the empty relation, `John` is so far the only element of the set `Male`. An explicit declaration of variables is not necessary. However, variables should be defined (i.e., assigned a value) before they are first used, otherwise CrocoPat prints a warning.

```
Male("Joe");
```

adds the string `Joe` to the set `Male`, such that it now has two elements. Similarly, we can initialize the variable `Female`:

```
Female("Alice");
Female("Jane");
Female("Mary");
```

To express that the John and Mary are the parents of Alice and Joe, and Joe is the father of Jane, we create a binary relation variable ParentOf which contains the five parent-child pairs:

```
ParentOf("John", "Alice");
ParentOf("John", "Joe");
ParentOf("Mary", "Alice");
ParentOf("Mary", "Joe");
ParentOf("Joe", "Jane");
```

Assignments. The following statement uses an *attribute* x to assign the set of Joe's parents to the set JoesParent:

```
JoesParent(x) := ParentOf(x, "Joe");
```

Now JoesParent contains the two elements John and Mary. As another example, the following assignment says that x is a child of y if and only if y is a parent of x:

```
ChildOf(x,y) := ParentOf(y,x);
```

John is the father of a person if and only if he is the parent of this person. The same is true for Joe:

```
FatherOf("John", x) := ParentOf("John", x);
FatherOf("Joe", x) := ParentOf("Joe", x);
```

Because the scope of each attribute is limited to one statement, the attribute in the first statement and the attribute in the second statement are different, despite of their equal name x.

Basic Relational Operators. The relation FatherOf can be described more concisely: x is father of y if and only if x is a parent of y and x is male:

```
FatherOf(x,y) := ParentOf(x,y) & Male(x);
```

Of course, we can define a similar relation for female parents:

```
MotherOf(x,y) := ParentOf(x,y) & Female(x);
```

Besides the operator *and* (&), another important operator is *or* (|). For example, we can define the ParentOf relation in terms of the relations MotherOf and FatherOf: x is a parent of y if and only if x is the mother or the father of y:

```
ParentOf(x,y) := MotherOf(x,y) | FatherOf(x,y);
```

Quantification. Parents are people who are a parent of another person. More precisely, x is a parent if and only if there *exists* (EX) a y such that x is a parent of y.

```
Parent(x) := EX(y, ParentOf(x,y));
```

Now the set Parent consists of John, Mary, and Joe. There is also an abbreviated notation for existential quantification which is similar to anonymous variables in Prolog and functional programming languages:

```
Parent(x) := ParentOf(x,_);
```

With the operator *not* (!), we can compute who has no children:

```
Childless(x) := !EX(y, ParentOf(x,y));
```

Equivalently, x childless if *for all* (FA) y holds that x is not a parent of y:

```
Childless(x) := FA(y, !ParentOf(x,y));
```

In both cases, the set Childless contains Alice and Jane.

Transitive Closure. To compute the grandparents of a person we have to determine the parents of his or her parents:

```
GrandparentOf(x,z) := EX(y, ParentOf(x,y) & ParentOf(y,z));
```

Now GrandparentOf contains the two pairs (John, Jane) and (Mary, Jane). To find out all ancestors of a person, i.e. parents, parents of parents, parents of parents of parents, etc., we have to apply the above operation (which is also called composition) repeatedly until the fixed point is reached, and unite the results. The transitive closure operator TC does exactly this:

```
AncestorOf(x,z) := TC(ParentOf(x,z));
```

The resulting relation AncestorOf contains any pair from ParentOf and GrandparentOf. (It also contains grand-grandparents etc., but there are none in this example.) The transitive closure operator TC can only be applied to binary relations.

Predefined Relations, the Universe. The relations FALSE and TRUE are predefined. FALSE is the empty relation, and TRUE is the full relation. More precisely, there is one predefined relation FALSE and one predefined relation TRUE for every arity. In particular, there is also a 0-ary relation FALSE(), which is the empty set, and a 0-ary relation TRUE(), which contains only () (the tuple of length 0). Intuitively, these 0-ary relations can be used like Boolean literals. By the way, the statement

```
Male("John");
```

is an abbreviation of the assignment

```
Male("John") := TRUE();
```

The result of TRUE(x) is the so-called *universe*. The universe contains all string literals that appear in the input RSF stream (if there is one, see Subsection 2.2) and on the left hand side of assignments in the present RML program. For example, the string literals used on the left hand side of the assignments in the examples of this subsection are Alice, Jane, Joe, John, and Mary, so the set TRUE(x) contains these five elements. See Subsection 3.4 for more information on the universe.

The binary relations =, !=, <, <=, >, and >= for the lexicographical order of the strings in the universe are also predefined. For example, siblings are two *different* people who have a common parent:

```
SiblingOf(x,y) := EX(z, ParentOf(z,x) & ParentOf(z,y)) & !=(x,y);
```

The infix notation is also available for binary relations, so the expression !=(x,y) can also be written as x!=y. Note that the predefined relations, like any other relation, are restricted to the universe. Thus the expression "A"="A" yields FALSE() if (and only if) the string A is not in the universe.

Further relational expressions are provided to match POSIX extended regular expressions [IEE01, Section 9.4]. These relational expressions start with the character @, followed by the string for the regular expression. For example,

```
StartsWithJ(x) := @"^J"(x);
```

assigns to the set StartsWithJ the set of all strings in the universe that start with the letter J, namely Jane, Joe, and John. A short overview of the syntax of regular expressions is given in Subsection 6.2.

Boolean Operators. Two relations can be compared with the operators =, !=, <, <=, >, or >=. Because such comparisons evaluate to either TRUE() or FALSE(), they are called Boolean expressions. For example,

```
GrandparentOf(x,y) < AncestorOf(x,y)
```

yields TRUE(), because GrandparentOf is a proper subset of AncestorOf. However,

```
GrandparentOf(x,y) = AncestorOf(x,y)
```

yields FALSE(), because the two relations are not equal.

The six comparison operators should not be confused with the six predefined relations for the lexicographical order. The operators take two relations as parameters, while the predefined relations take strings or attributes as parameters.

2.2 Input and Output of Relations

File Format RSF. CrocoPat reads and writes relations in Rigi Standard Format (RSF, [Won98, Section 4.7.1]). Files in RSF are human-readable, can be loaded into and saved from many reverse engineering tools, and are easily processed by scripts in common scripting languages.

In an RSF file, a tuple of an n-ary relation is represented as a line of the form

```
RelationName element1 element2 ... elementn
```

The elements may be enclosed by double quotes. Because white space serves as delimiter of the elements, elements that contain white space *must* be enclosed by double quotes. A relation is represented as a sequence of such lines. The order of the lines is arbitrary. An RSF file may contain several relations.

As an example, the relation `ParentOf` from the previous subsection can be represented in RSF format as follows:

```
ParentOf John Alice
ParentOf John Joe
ParentOf Mary Alice
ParentOf Mary Joe
ParentOf Joe  Jane
```

Input. RML has no input statements. When CrocoPat is started, it first reads input relations in RSF from the standard input before it parses and executes the RML program. RSF reading can be skipped with the `-e` command line option. If the input relations are available as files, they can be feeded into CrocoPat's standard input using the shell operator `<`, as the following examples shows for the file `ParentOf.rsf`:

```
crocopat Prog.rml < ParentOf.rsf
```

The end of the input data is recognized either from the end of file character or from a line that starts with the dot (`.`) character. The latter is sometimes useful if RSF input is feeded interactively.

If the above RSF data is used as input, then at the start of the program the binary relation variable `ParentOf` contains the five pairs, and the universe contains the five string literals `Alice`, `Jane`, `Joe`, `John`, and `Mary` (and additionally all string literals that appear on the left hand side of assignments in the program.)

Output. The `PRINT` statement outputs relations in RSF format to the standard output. For example, running the program

```
ParentOf("Joe",x) := FALSE(x);
ParentOf(x,"Joe") := FALSE(x);
PRINT ParentOf(x,y);
```

with the above input data prints to the standard output

```
John Alice
Mary Alice
```

The statement

```
PRINT ["ParentOf"] ParentOf(x,y);
```

writes the string `ParentOf` before each tuple, and thus outputs

```
ParentOf John Alice
ParentOf Mary Alice
```

The output can also be appended to a file `ParentOf2.rsf` (which is created if it does not exist) with

```
PRINT ["ParentOf"] ParentOf(x,y) TO "ParentOf2.rsf";
```

or to stderr with

```
PRINT ["ParentOf"] ParentOf(x,y) TO STDERR;
```

Command Line Arguments. It is sometimes convenient to specify the names of output files at the command line and not in the RML program. If there is only one output file, the standard output can be simply redirected to a file using the shell operator `>`:

```
crocopat Prog.rml < ParentOf.rsf > ParentOf2.rsf
```

An alternative solution (which also works with more than one file) is to pass command line arguments to the program. Command line arguments can be accessed in RML as `$1`, `$2`, etc. For example, when the program

```
ChildOf(x,y) := ParentOf(y,x);
PRINT ["Child"] ChildOf(x,$1) TO $1 + ".rsf";
PRINT ["Child"] ChildOf(x,$2) TO $2 + ".rsf";
```

is executed with

```
crocopat IO.rml Joe Mary < ParentOf.rsf
```

then the first `PRINT` statement writes to the file `Joe.rsf`, and the second `PRINT` statement writes to `Mary.rsf`.

Command line arguments are not restricted to specifying file names, but can be used like string literals. However, in contrast to string literals, command line arguments are never added to the universe, and thus cannot be used on the left hand side of relational assignments.

2.3 Control Structures

This subsection introduces the control structures of RML, using algorithms for computing the transitive closure of a binary relation `R` as examples.

`WHILE` **Statement.** As a first algorithm, the relation `R` is composed with itself until the fixed point is reached.

```
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) := Result(x,z) | EX(y, Result(x,y) & Result(y,z));
}
```

The program illustrates the use of the `WHILE` loop, which has the usual meaning: The body of the loop is executed repeatedly as long as the condition after `WHILE` evaluates to `TRUE()`.

`FOR` **Statement.** The second program computes the transitive closure of the relation `R` using the Warshall algorithm. This algorithm successively adds arcs. In the first iteration, an arc (u, v) is added if the input graph contains the arcs (u, node_0) and (node_0, v). In the second iteration, an arc (u, v) is added if the graph that results from the first iteration contains the arcs (u, node_1) and (node_1, v). And so on, for all nodes of the graph (in arbitrary order.)

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

The program illustrates the use of the `FOR` loop. The relation after `IN` must be a unary relation. The iterator after `FOR` is a string variable and takes as values the elements of the unary relation in lexicographical order. Thus, the number of iterations equals the number of elements of the unary relation.

For the implementation of the transitive closure operator of RML, we experimented with several algorithms. An interesting observation in these experiments was that the empirical complexity of some

algorithms for practical graphs deviated strongly from their theoretical worst case complexity, thus some algorithms with a relatively bad worst-case complexity were very competitive in practice. In our experiments, the first of the above algorithms was very fast, thus we made it available as operator `TCFAST`. The implementation of the `TC` operator of RML is a variant of the Warshall algorithm. It is somewhat slower than `TCFAST` (typically about 20 percent in our experiments), but often needs much less memory because it uses no ternary relations.

`IF` Statement. The following example program determines if the input graph `R` is acyclic, by checking if its transitive closure contains loops (i.e. arcs from a node to itself):

```
SelfArcs(x,y) := TC(R(x,y)) & (x = y);
IF (SelfArcs(_,_)) {
    PRINT "R is not acyclic", ENDL;
} ELSE {
    PRINT "R is acyclic", ENDL;
}
```

2.4 Relations of Higher Arity

In this subsection, relations of arity greater than two are used for finding potential design patterns and design problems in structural models of object-oriented programs. The examples are taken from [BNL03].

The models of object-oriented programs contain the call, containment, and inheritance relationships between classes. Here containment means that a class has an attribute whose type is another class. The direction of inheritance relationships is from the subclass to the superclass. As an example, the source code

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

corresponds to the following RSF file:

```
Inherit   SubClass  SuperClass
Contain   SubClass  ContainedClass
```

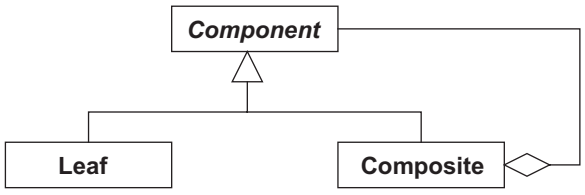


Fig. 1. Composite design pattern

Composite Design Pattern. Figure 1 shows the class diagram of the Composite design pattern [GHJV95]. 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) Composite and Leaf are subclasses of Component, (2) Composite contains an instance of Component, and (3) Leaf does not contain an instance of Component. The translation of these conditions to an RML statement is straightforward:

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

Degenerate Inheritance. When a class `C` inherits from another class `A` directly and indirectly via a class `B`, the direct inheritance is probably redundant or even misleading. The following statement detects such patterns:

```
DegInh(a,b,c) :=   Inherit(c,b)
                 & Inherit(c,a)
                 & TC(Inherit(b,a));
```

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 this class is difficult. All classes that participate in cycles can be found using the transitive closure operator, as shown in Subsection 2.3. However, in many large software systems hundreds of classes participate in cycles, and it is tedious for a human analyst to find the actual cycles in the list of these classes. In our experience, it is often more useful to detect cycles in the order of ascending length. As a part of such a program, the following statements detects all cycles of length 3.

```
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);
```

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`).

2.5 Numerical Expressions

In this subsection, a software metric is calculated as example for the use of numerical expressions in RML programs. Therefore, we extend the structural model of object-oriented programs introduced in the previous subsection with a binary relation `PackageOf`. This relation assigns to each package the classes that it contains. (Packages are high-level entities in object-oriented software systems that can be considered as sets of classes.)

Robert Martin's metric for the instability of a package is defined as $ce/(ca + ce)$, where ca is the number of classes outside the package that use classes inside the package, and ce is the number of classes inside the package that use classes outside the package [Mar97].

```
Use(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
Package(x) := PackageOf(x,_);
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));
    IF (ca + ce > 0) {
        PRINT p, " ", ce / (ca+ce), ENDL;
    }
}
```

3 Advanced Programming Techniques

This section describes advanced programming techniques, in particular for improving efficiency and circumventing language limitations. The first subsection explains how to control the memory usage of CrocoPat. The second and third subsection describe how relational expressions are evaluated in CrocoPat, and how to assess and improve the efficiency of their evaluation. The fourth subsection explains why the universe is immutable during the execution of an RML program and how to work around this limitation.

3.1 Controlling the Memory Usage

CrocoPat represents relations using the data structure binary decision diagram (BDD, [Bry86]). When CrocoPat is started, it reserves a fixed amount of memory for BDDs, which is not changed during the execution of the RML program. If the available memory is insufficient, CrocoPat exits with the error message

```
Error: BDD package out of memory.
```

The BDD memory can be controlled with the command line option `-m`, followed by an integer number giving the amount of memory in MByte. The default value is 50. The actual amount of memory reserved for BDDs is not infinitely variable, so the specified value is only a rough upper bound of the amount of memory used.

It can also be beneficial to reserve less memory, because the time used for allocating memory increases with the amount of memory. When the manipulated relations are small or the algorithms are computationally inexpensive, memory allocation can dominate the overall runtime.

3.2 Speeding up the Evaluation of Relational Expressions

This subsection explains how CrocoPat evaluates relational expressions. Based on this information, hints for performance improvement are given. Understanding the subsection requires basic knowledge about BDDs and the impacts of the variable order on the size of BDDs. An introduction to BDDs is beyond the scope of this manual, we refer the reader to [Bry92].

The attributes in an RML program are called *user attributes* in the following. For example, the expression `R(x,y)` contains the user attributes `x` and `y`. For the internal representation of relations, CrocoPat uses a sequence of *internal attributes*, which are distinct from the user attributes. We call these internal attributes `i1`, `i2`, `i3`, ... For example, the binary relation `R` is internally represented as a set of assignments to the internal attributes `i1` and `i2`.

When the expression `R(x,y)` is evaluated, the internal attributes `i1` and `i2` are renamed to the user attributes `x` and `y`. Therefore all BDD nodes of the representation of `R` have to be traversed. Thus, the time for evaluating the expression `R(x,y)` is at least linear in the number of BDD nodes of `R`'s representation.

The order of the internal attributes in the BDD is always `i1`, `i2`, ... The order of the user attributes in the BDD may be different in the evaluation of each statement, because the scope of user attributes is restricted to one statement. The order of the user attributes in the BDD in the evaluation of a statement is the order in which CrocoPat encounters the user attributes in the execution of the statement. In the example statement

```
R(x,z) := EX(y, R(x,y) & R(y,z));
```

CrocoPat evaluates first `R(x,y)`, then `R(y,z)`, then the conjunction, then the existential quantification, and finally the assignment. Therefore, the order of the user attributes in the BDD is `x`, `y`, `z`.

Avoid renaming large relations. The time for the evaluation of the expression `R(x,y)` is at least linear in the number of BDD nodes in the representation of `R`, because all BDD nodes have to be renamed from internal attributes to user attributes. Usually this effort for renaming does not dominate the overall runtime, but in the following we give an example where it does.

Let `R(x,y)` be a directed graph with n nodes. Let the BDD representation of `R` have $\Theta(n^2)$ BDD nodes (which is the worst case). The assignment

```
Outneighbor(y) := EX(x, R(x,y) & x="node1");
```

assigns the outneighbors of the graph node `node1` to the set `Outneighbor`. The evaluation of `R(x,y)` costs $\Theta(n^2)$ time in this example, because of the renaming of all nodes. The "real computation", namely the conjunction and the existential quantification, can be done in $O(\log n)$ time. So the renaming dominates the overall time.

The equivalent statement

```
Outneighbor(y) := R("node1",y);
```

is executed in only $O(n)$ time, because the set `R("node1",y)` has $O(n)$ elements, and its BDD representation has $O(n)$ nodes.

Avoid swapping attributes. Renaming the nodes of a BDD costs at least linear time, but can be much more expensive when attributes have to be swapped. In the statement

```
S(x,y,z) := R(y,z) & R(x,y);
```

the BDD attribute order on the right hand side of the assignment is `y`, `z`, `x`, while the BDD attribute order on the left hand side is `x`, `y`, `z`. Because the two orders are different, attributes have to be swapped to execute the assignment. This can be easily avoided by using the equivalent statement

```
S(x,y,z) := R(x,y) & R(y,z);
```

Of course, swapping attributes can not always be avoided. However, developers of RML programs should know that swapping attributes can be expensive, and should minimize it when performance is critical.

Ensure good attribute orders. A detailed discussion of BDD attribute orders is beyond the scope of this manual (see e.g. [Bry92, Section 1.3] for details), but the basic rule is that related attributes should be grouped together. In the two assignment statements

```
S1(v,w,x,y) := R(v,w) & R(x,y);
S2(v,x,w,y) := R(v,w) & R(x,y);
```

the attributes `v` and `w` are related, and the attributes `x` and `y` are related, while `v` and `w` are unrelated to `x` and `y`. In `S1`, related attributes are grouped together, but not in `S2`. For many relations `R`, the BDD representation of `S1` will be drastically smaller than the BDD representation of `S2`.

Profile. Information about the number of BDD nodes and the BDD attribute order of an expression can be printed with `PRINT RELINFO`. For example,

```
PRINT RELINFO(R(y,z) & R(x,y));
```

may output

```
Number of tuples in the relation: 461705
Number of values (universe): 6218
Number of BDD nodes: 246986
Percentage of free nodes in BDD package: 1614430 / 1966102 = 82 %
Attribute order: y z x
```

The first line gives the cardinality of the relation, the second line the cardinality of the universe, the third line the size of the BDD that represents the result of the expression, and the fifth line the attribute order in this BDD.

3.3 Estimating the Evaluation Time of Relational Expressions

Knowledge of the computational complexity of RML's operators is useful to optimize the performance of RML programs. This subsection gives theoretical complexity results, but also discusses the limits of their practical application.

Table 1 shows the asymptotic worst case time complexity for the evaluation of RML's relational operators. The times do not include the renaming of internal attributes discussed in the previous subsection, and the evaluation of subexpressions. It is assumed that the caches of the BDD package are sufficiently large. This assumption is closely approximated in practice when the manipulated BDDs only occupy a small fraction of the available nodes in the BDD package. Otherwise, performance may be improved by increasing the BDD memory (see Subsection 3.1).

When the operands of an expression are relations, the computation time is given as function of the sizes of their BDD representation. (The only exception are the transitive closure operators, where a function of the size of the universe gives a more useful bound.) This raises the problem of how to estimate these BDD sizes. Many practical relations have regularities that enable an (often dramatically) compressed BDD representation, but the analytical derivation of the typical compression rate for relations from a particular application domain is generally difficult. Our advice is to choose some representative examples and measure the BDD sizes with the `PRINT RELINFO` statement.

It is important to note that Table 1 gives *worst-case* computation times. In many cases, the typical practical performance is much better than the worst case. For example, the relational comparison operators (`<=`, `<`, `>=`, `>`) and the binary logic operators (`&`, `|`, `->`, `<->`) are very common in RML programs. Their worst-case complexity is the product of the sizes of their operand BDDs, which is alarmingly high. However, in practice the performance is often much closer to the sum of the operand BDD sizes. Similarly, the quantification operators are often efficient despite their prohibitive worst case runtime (which is difficult to derive because quantification is implemented as a series of several bit-level operations).

Another practically important example for the gap between average-case and worst-case runtime are the transitive closure operators. The worst case complexity of their BDD-based implementation is the same as for implementations with conventional data structures. However, the BDD-based implementations are much more efficient for many practical graphs [BNL03]. Even in the comparison of different BDD-based implementations, a better worst-case complexity does not imply a better performance in practice. We conclude from our experience that knowledge of the theoretical complexity complements but cannot replace experimentation in the development of highly optimized RML programs.

`! re`	$O(bddsize(\texttt{re}))$
`re1 & re2, re1 | re2`	$O(bddsize(\texttt{re1}) \cdot bddsize(\texttt{re2}))$
`re1 -> re2, re1 <-> re2`	$O(bddsize(\texttt{re1}) \cdot bddsize(\texttt{re2}))$
`EX(x, re), FA(x, re)`	$\geq O(bddsize(\texttt{re})^2)$
`TC(re)`	$O(n^3)$
`TCFAST(re)`	$O(n^3 \log n)$
`FALSE(x1, x2, ...)`	$O(1)$
`TRUE(x1, x2, ...)`	$O(1)$
`@s(x)`	$O(n \log n)$
`~(x1, x2)`	$O(n \log n)$ (`~` can be `=`, `!=`, `<=`, `<`, `>=`, `>`)
`~(ne1, ne2)`	$O(1)$ (`~` can be `=`, `!=`, `<=`, `<`, `>=`, `>`)
`re1 = re2, re1 != re2`	$O(1)$
`re1 < re2, re1 <= re2`	$O(bddsize(\texttt{re1}) \cdot bddsize(\texttt{re2}))$
`re1 > re2, re1 >= re2`	$O(bddsize(\texttt{re1}) \cdot bddsize(\texttt{re2}))$

Table 1. Worst case time complexity of the evaluation of relational expressions. `re`, `re1`, `re2` are relational expressions, `x`, `x1`, `x2` are attributes, and `ne1`, `ne2` are numerical expressions. $bddsize(\texttt{re})$ is the number of BDD nodes of the result of the expression `re`, and n is the cardinality of the universe.

3.4 Extending the Universe

The set of all strings that may be tuple elements of relations in an RML program is called the *universe*. The universe contains all tuple elements of the input relations (from the input RSF data), and all string literals that appear on the left hand side of assignments in the RML program. The universe is immutable in the sense that it can be determined before the interpretation of the RML program starts, and is not changed during the interpretation of the RML program.

Sometimes the immutability of the universe is inconvenient for the developer of RML programs. Consider, for example, a program that takes as input the nodes and arcs of a graph, and computes the binary relation `OutneighborCnt` which contains for each node the number of outneighbors:

```
FOR n IN Node(x) {
    OutneighborCnt(n, #(Arc(n,x))) := TRUE();
}
```

This is not a syntactically correct RML program, because `#(Arc(n,x))` is not a string, but a number. However, RML has an operator `STRING` that converts a number into a string. But

```
OutneighborCnt(n, STRING( #(Arc(n,x)) )) := TRUE();
```

is still not syntactically correct, because such a conversion is not allowed at the left hand side of assignment statements. The reason is that the string that results from such a conversion is generally not known before the execution of the RML program, can therefore not be added to the universe before the execution, and is thus not allowed as tuple element of a relation.

The immutability of the universe during the execution of an RML program is necessary because constant relations like `TRUE(x)` (the universe) and `=(x,y)` (string equality for all strings in the universe) are only defined for a given universe. Also, the complement of a relation depends on the universe: The complement of a set contains all strings of the universe that are not in the given set, and thus clearly changes when the universe changes.

However, there is a way to work around this limitation: Writing an RSF file, and restarting CrocoPat with this RSF file as input, which adds all tuple elements in the RSF file to the universe. For example, the above incorrect program can be replaced by the following correct program:

```
FOR n IN Node(x) {
    PRINT "OutneighborCnt ", n, " ", #(Arc(n,x)) TO "OutneighborCnt.rsf";
}
```

When CrocoPat is restarted with the resulting RSF file `OutneighborCnt.rsf` as input, the binary relation `OutneighborCnt` is available for further processing.

4 CrocoPat Reference

CrocoPat is executed with

```
crocopat [OPTION]... FILE [ARGUMENT]...
```

It first reads relations in RSF (see Section 5) from stdin (unless the option `-e` is given) and then executes the RML program `FILE` (see Section 6). The `ARGUMENT`s are passed to the RML program. The `OPTION`s are

`-e`	Do not read RSF data from stdin.
`-m NUMBER`	Approximate memory for BDD package in MB. The default is 50. See Subsection 3.1.
`-q`	Suppress warnings.
`-h`	Display help message and exit.
`-v`	Print version information and exit.

The output of the RML program can be written to files, stdout, or stderr, as specified in the RML program. Error messages and warnings of CrocoPat are always written to stderr.

The exit status of CrocoPat is 1 if it terminates abnormally and 0 otherwise. CrocoPat always outputs an error message to stderr before it terminates with exit status 1.

5 RSF Reference

Rigi Standard Format (RSF) is CrocoPat's input and output format for relations. It is an extension of the format for binary relations defined in [Won98, Section 4.7.1]. For examples of its use see Subsection 2.2.

An RSF stream is a sequence of lines. The order of the lines is arbitrary. The repeated occurrence of a line is permissable and has the same meaning as a single occurrence. The end of an RSF stream is indicated by the end of the file or by a line that starts with a dot (`.`). Lines starting with a sharp (`#`) are comment lines.

All lines that do not start with a dot or a sharp specify a tuple in a relation. They consist of the name of the relation followed by a sequence of (an arbitrary number of) tuple elements, separated by at least one whitespace character (i.e., space or horizontal tab).

Relation names must be RML identifiers (see Subsection 6.1). Tuple elements are sequences of arbitrary characters except line breaks and whitespace characters. A tuple element may be optionally enclosed by double quotes (`"`), in which case it may also contain whitespace characters. Tuple elements that are enclosed by double quotes in the RSF input of an RML program are also enclosed by double quotes in its output.

6 RML Reference

Relation Manipulation Language (RML) is CrocoPat's programming language for manipulating relations. This section defines the lexical structure, the syntax, and the semantics of RML. Nonterminals are printed in italics and terminals in typewriter.

6.1 Lexical Structure

Identifiers are sequences of Latin letters (`a-zA-z`), digits (`0-9`) and underscores (`_`), the first of which must be a letter or underscore. RML has four types of identifiers: attributes (*attribute*), relational variables (*rel_var*), string variables (*str_var*), and numerical variables (*num_var*). Every identifier of an RML program belongs to exactly one of these types. The type is determined at the first occurrence of the identifier in the input RSF file (only possible for relational variables) or in the RML program. Explicit declaration of identifiers is not necessary.

The following strings are reserved as keywords and therefore cannot be used as identifiers:

```
AVG DIV ELSE ENDL EX EXEC EXIT FA FOR IF IN MAX MIN MOD NUMBER PRINT RELINFO
STDERR STRING SUM TC TCFAST TO WHILE
```

RML has two types of literals: string literals (*str_literal*) and numerical literals (*num_literal*). String literals are delimited by double quotes (`"`) and can contain arbitrary characters except double quotes. A numerical literal consists of an integer part, a fractional part indicated by a decimal point (`.`), and an exponent indicated by the letter `e` or `E` followed by an optionally signed integer. All three parts are optional, but at least one digit in the integer part or the fractional part is required. Examples of numerical literals are `1`, `.2`, `3.`, `4.5`, and `6e-7`.

There are two kinds of comments: Text starting with `/*` and ending with `*/`, and text from `//` to the end of the line.

6.2 Syntax and Informal Semantics

program ::=	**RML program.**
stmt ...[1]	Executes the *stmt*s in the given order.
stmt ::=	**Statement.**
rel_var(*term*,...) := *rel_expr*;	Assigns the result of *rel_expr* to *rel_var*.
	→[2]The *term*s must be *attribute*s or *str_literal*s.
	→ The set of *attribute*s among the *term*s on the left hand side must equal the set of free attributes in *rel_expr*.
rel_var(*term*,...);	Shortcut for *rel_var*(*term*,...) := TRUE(*term*,...).
str_var := *str_expr*;	Assigns the result of *str_expr* to *str_var*.
num_var := *num_expr*;	Assigns the result of *num_expr* to *num_var*.
IF *rel_expr* {*stmt* ...} ELSE {*stmt* ...}	Executes the *stmt*s before ELSE if the result of *rel_expr* is TRUE(),
IF *rel_expr* {*stmt* ...}	and the *stmt*s after ELSE (if present) otherwise.
	→ *rel_expr* must not have free attributes.
WHILE *rel_expr* {*stmt* ...}	Exec. the *stmt*s repeatedly as long as *rel_expr* evaluates to TRUE().
	→ *rel_expr* must not have free attributes.
FOR *str_var* IN *rel_expr* {*stmt* ...}	Executes the *stmt*s once for each element in the result of *rel_expr*.
	→ *rel_expr* must have exactly one free attribute.
PRINT *print_expr*,...;	Writes the results of the *print_expr*s to stdout.
PRINT *print_expr*,... TO STDERR;	Writes the results of the *print_expr*s to stderr.
PRINT *print_expr*,... TO *str_expr*;	Appends the results of the *print_expr*s to the specified file.
EXEC *str_expr*;	Executes the shell command given by *str_expr*.
	The exit status is available as numerical constant exitStatus.
EXIT *num_expr*;	Exits CrocoPat with the given exit status.
{ *stmt* ... }	Executes the *stmt*s in the given order.
rel_expr ::=	**Relational Expression. The result is a relation.**
rel_var(*term*,...)	Atomic relational expression.
term *rel_var* *term*	Same as *rel_var*(*term*, *term*).
! *rel_expr*	Negation (not).
rel_expr & *rel_expr*	Conjunction (and).
rel_expr | *rel_expr*	Disjunction (or).
rel_expr -> *rel_expr*	Implication (if). r1 -> r2 is equivalent to !(r1) | (r2).
rel_expr <-> *rel_expr*	Equivalence (if and only if).
	r1 <-> r2 is equivalent to (r1 -> r2) & (r2 -> r1).
EX(*attribute*,..., *rel_expr*)	Existential quantification of the *attribute*s.
FA(*attribute*,..., *rel_expr*)	Universal quantification of the *attribute*s.
TC(*rel_expr*)	Transitive closure.
	→ *rel_expr* must have exactly two free attributes.
TCFAST(*rel_expr*)	Same as TC, but with an alternative algorithm (see Section 2.3).
FALSE(*term*,...)	Empty relation.
TRUE(*term*,...)	Relation containing all tuples of strings in the universe.
@*str_expr*(*term*)	Strings in the universe that match the regular expression *str_expr*.
~(*term*, *term*)	Lexicographical order of all strings in the universe.[3]
term ~ *term*	Same as ~(*term*, *term*).
~(*num_expr*, *num_expr*)	Numerical comparison. The result is either TRUE() or FALSE().[3]
num_expr ~ *num_expr*	Same as ~(*num_expr*, *num_expr*).
rel_expr ~ *rel_expr*	Relational comparison. The result is either TRUE() or FALSE().[3]
(*rel_expr*)	

[1] *stmt* ... denotes a sequence of one or more *stmt*s.
[2] Context conditions are marked with →.
[3] ~ can be =, !=, <=, <, >=, >.

term ::=	**Term.**
attribute	Attribute.
_	Anonymous attribute. E.g. `R(_)` is equivalent to `EX(x, R(x))`.
str_expr	String expression.
str_expr ::=	**String Expression. The result is a string.**
str_literal	String literal.
str_var	String variable.
`STRING(`*num_expr*`)`	Converts the result of *num_expr* into a string.
`$` *num_expr*	Command line argument. The first argument has the number 1. The constant `argCount` contains the number of arguments.
str_expr `+` *str_expr*	Concatenation.
`(`*str_expr*`)`	
num_expr ::=	**Numerical Expression. The result is a floating point number.**
num_literal	Numerical literal.
num_var	Numerical variable.
`NUMBER(`*str_expr*`)`	Converts the result of *str_expr* into a number. Yields 0.0 if the result of *str_expr* is not the string representation of a number.
`#(`*rel_expr*`)`	Cardinality (number of elements) of the result of *rel_expr*.
`MIN(`*rel_expr*`)`, `MAX(`*rel_expr*`)`, `SUM(`*rel_expr*`)`, `AVG(`*rel_expr*`)`	Minimum, maximum, sum, and arithmetic mean of `NUMBER(s)` over all strings `s` in the result of *rel_expr*. → *rel_expr* must have one free attribute, its result must be non-empty.
num_expr `+` *num_expr*	Addition.
num_expr `-` *num_expr*	Subtraction.
num_expr `*` *num_expr*	Multiplication.
num_expr `/` *num_expr*	Real division.
num_expr `DIV` *num_expr*	Integer division (truncating).
num_expr `MOD` *num_expr*	Modulo.
num_expr `^` *num_expr*	The first *num_expr* raised to the power given by the second *num_expr*.
`argCount`	Number of command line arguments.
`exitStatus`	Exit status of the last executed `EXEC` statement.
`(`*num_expr*`)`	
print_expr ::=	**Print Expression.**
rel_expr	Prints the tuples in the result of *rel_expr*, one tuple per line.
`[`*str_expr*`]` *rel_expr*	Prints the result of *str_expr* before each tuple of *rel_expr*.
str_expr	Prints the result of *str_expr*.
num_expr	Prints the result of *num_expr*.
`ENDL`	Prints a line break.
`RELINFO(`*rel_expr*`)`	Prints information about the BDD representation of *rel_expr*.

Levels of Precedence (from Low to High)

`=`, `!=`, `<=`, `<`, `>=`, `>`
`->`, `<->`
`|`
`&`
`!`
`+`, `-` (binary)
`*`, `/`, `DIV`, `MOD`
`^`
`-` (unary)
`$`

Free Attributes in Relational Expressions. Several context conditions of RML refer to the free attributes in relational expressions. The number of free attributes in a relational expression equals the arity of the resulting relation. Informally, the set of free attributes of an expression is the set of its contained attributes that are not in the scope of a quantifier (i.e., `EX` or `FA`). Exceptions are the numerical and relational comparison, which have Boolean results and therefore no free attributes. Formally, the function f that assigns to each relational expression the set of its free attributes is inductively defined as follows:

$f($ *rel_var*`(`*term*$_1$`,` *term*$_2$`,...)` $) := \{$ *term*$_i$ $\mid$ *term*$_i$ is an attribute $\}$
$f($ `!`*rel_expr* $) := f($ *rel_expr* $)$
$f($ *rel_expr*$_1$ `&` *rel_expr*$_2$ $) := f($ *rel_expr*$_1$ $) \cup f($ *rel_expr*$_2$ $)$
$f($ *rel_expr*$_1$ `|` *rel_expr*$_2$ $) := f($ *rel_expr*$_1$ $) \cup f($ *rel_expr*$_2$ $)$
$f($ *rel_expr*$_1$ `->` *rel_expr*$_2$ $) := f($ *rel_expr*$_1$ $) \cup f($ *rel_expr*$_2$ $)$
$f($ *rel_expr*$_1$ `<->` *rel_expr*$_2$ $) := f($ *rel_expr*$_1$ $) \cup f($ *rel_expr*$_2$ $)$
$f($ `EX(`*attribute*`,` *rel_expr*`)` $) := f($ *rel_expr* $) \setminus \{$ *attribute* $\}$
$f($ `FA(`*attribute*`,` *rel_expr*`)` $) := f($ *rel_expr* $) \setminus \{$ *attribute* $\}$
$f($ `TC(`*rel_expr*`)` $) := f($ *rel_expr* $)$
$f($ `TCFAST(`*rel_expr*`)` $) := f($ *rel_expr* $)$
$f($ `(`*num_expr*$_1$ `~` *num_expr*$_2$`)` $) := \emptyset$
$f($ `(`*rel_expr*$_1$ `~` *rel_expr*$_2$`)` $) := \emptyset$
$f($ `(`*rel_expr*`)` $) := f($ *rel_expr* $)$

As before, `~` can be `=`, `!=`, `<=`, `<`, `>=`, or `>`. The relational constants `~`, `FALSE`, `TRUE`, and `@`*str_expr* are equivalent to *rel_var* with respect to the definition of free attributes.

Regular Expressions. In the relational expression `@`*str_expr*`(`*term*`)`, the result of *str_expr* can be any POSIX extended regular expression [IEE01, Section 9.4]. A full description is beyond the scope, we only give a short overview.

Most characters in a regular expression only match themselves. The following special characters match themselves only when they are preceded by a backslash (`\`), and otherwise have special meanings:

`.`	Matches any single character.
`[ ]`	Matches any single character contained within the brackets.
`[^ ]`	Matches any single character not contained within the brackets.
`^`	Matches the start of the string.
`$`	Matches the end of the string.
$\{x$`,`$y\}$	Matches the last character (or regular expression enclosed by parentheses) at least x and at most y times.
`+`	Matches the last character (or regular expression enclosed by parentheses) one or more times.
`*`	Matches the last character (or regular expression enclosed by parentheses) zero or more times.
`?`	Matches the last character (or regular expression enclosed by parentheses) zero or one times.
`|`	Matches either the expression before or the expression after the operator.

Regular expressions can be grouped by enclosing them with parentheses (`(...)`).

6.3 Formal Semantics

This subsection formally defines the semantics of the part of RML that deals with relations, namely of relational expressions and the relational assignment statement.

The *universe* $\mathcal{U}$ is the finite set of all string literals that appear in the input RSF file, or on the left side of a relational assignment. The finite set of *attributes* of the RML program is denoted by $\mathcal{X}$ ($\mathcal{U} \cap \mathcal{X} = \emptyset$). An *attribute assignment* is a total function $v : \mathcal{X} \cup \mathcal{U} \to \mathcal{U}$ which maps each attribute to its value and (for notational convenience) each string literal to itself. The set of all attribute assignments is denoted by $Val(\mathcal{X})$.

The finite set of *relation variables* in the RML program is denoted by $\mathcal{R}$. A *relation assignment* is a total function $s : \mathcal{R} \to \bigcup_{n\in\mathbb{N}} 2^{\prod_{k=1}^{n} \mathcal{U}}$, which maps each relation variable to a relation of arbitrary arity. The set of all relation assignments is denoted by $Rel(\mathcal{R})$.

The semantics of relational expressions and statements are given by the following interpretation functions:

$$[\![.]\!]_e : rel\_exprs \to (Rel(\mathcal{R}) \to 2^{Val(\mathcal{X})})$$
$$[\![.]\!]_s : stmts \to (Rel(\mathcal{R}) \to Rel(\mathcal{R}))$$

So we define the semantics of an expression as the set of attribute assignments that satisfy the expression, and the semantics of a statement as a transformation of the relation assignment. The interpretation functions are defined inductively in Figure 2.

$$\begin{aligned}
[\![rel_var\texttt{(}term_1\texttt{, ..., }term_n\texttt{)}]\!]_e(s) &= \Big\{ v \in Val(\mathcal{X}) \;\Big|\; \big(v(term_1), \ldots, v(term_n)\big) \in s(rel_var) \Big\} \\
[\![\texttt{! }rel_expr]\!]_e(s) &= Val(\mathcal{X}) \setminus [\![rel_expr]\!]_e(s) \\
[\![rel_expr_1 \texttt{ \& } rel_expr_2]\!]_e(s) &= [\![rel_expr_1]\!]_e(s) \cap [\![rel_expr_2]\!]_e(s) \\
[\![rel_expr_1 \texttt{ | } rel_expr_2]\!]_e(s) &= [\![rel_expr_1]\!]_e(s) \cup [\![rel_expr_2]\!]_e(s) \\
[\![rel_expr_1 \texttt{ -> } rel_expr_2]\!]_e(s) &= [\![\texttt{! (}rel_expr_1\texttt{) | (}rel_expr_2\texttt{)}]\!]_e(s) \\
[\![rel_expr_1 \texttt{ <-> } rel_expr_2]\!]_e(s) &= [\![\texttt{(}rel_expr_1 \texttt{ -> } rel_expr_2\texttt{) \& (}rel_expr_2 \texttt{ -> } rel_expr_1\texttt{)}]\!]_e(s) \\
[\![\texttt{EX(}attribute\texttt{, }rel_expr\texttt{)}]\!]_e(s) &= \Big\{ v \in Val(\mathcal{X}) \;\Big|\; \exists v' \in [\![rel_expr]\!]_e(s)\ \forall x \in \mathcal{X} \setminus \{attribute\} : v(x) = v'(x) \Big\} \\
[\![\texttt{FA(}attribute\texttt{, }rel_expr\texttt{)}]\!]_e(s) &= [\![\texttt{! EX(}attribute\texttt{, ! (}rel_expr\texttt{))}]\!]_e(s) \\
[\![\texttt{TC(}rel_var\texttt{(}attribute_1\texttt{, }attribute_2\texttt{))}]\!]_e(s) &\overset{lfp}{=} [\![\quad rel_var\texttt{(}attribute_1\texttt{, }attribute_2\texttt{)} \\
&\qquad \texttt{| EX(}attribute_3\texttt{, }\quad rel_var\texttt{(}attribute_1\texttt{, }attribute_3\texttt{)} \\
&\qquad\qquad \texttt{\& TC(}rel_var\texttt{(}attribute_3\texttt{, }attribute_2\texttt{)))}]\!]_e(s) \\
[\![\texttt{TCFAST(}rel_var\texttt{(}attribute_1\texttt{, }attribute_2\texttt{))}]\!]_e(s) &= [\![\texttt{TC(}rel_var\texttt{(}attribute_1\texttt{, }attribute_2\texttt{))}]\!]_e(s)
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{TRUE(}term_1\texttt{, ..., }term_n\texttt{)}]\!]_e(s) &= Val(\mathcal{X}) \\
[\![\texttt{FALSE(}term_1\texttt{, ..., }term_n\texttt{)}]\!]_e(s) &= \emptyset \\
[\![\texttt{=(}term_1\texttt{, }term_2\texttt{)}]\!]_e(s) &= \Big\{ v \in Val(\mathcal{X}) \mid v(term_1) = v(term_2) \Big\} \\
[\![\texttt{<(}term_1\texttt{, }term_2\texttt{)}]\!]_e(s) &= \Big\{ v \in Val(\mathcal{X}) \mid v(term_1) <_{\text{lexicographically}} v(term_2) \Big\}
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{=(}rel_expr_1\texttt{, }rel_expr_2\texttt{)}]\!]_e(s) &= \begin{cases} Val(\mathcal{X}), & \text{if } [\![rel_expr_1]\!]_e(s) = [\![rel_expr_2]\!]_e(s) \\ \emptyset, & \text{otherwise} \end{cases} \\
[\![\texttt{<(}rel_expr_1\texttt{, }rel_expr_2\texttt{)}]\!]_e(s) &= \begin{cases} Val(\mathcal{X}), & \text{if } [\![rel_expr_1]\!]_e(s) \subsetneq [\![rel_expr_2]\!]_e(s) \\ \emptyset, & \text{otherwise} \end{cases} \\
[\![\texttt{(}rel_expr\texttt{)}]\!]_e(s) &= [\![rel_expr]\!]_e(s)
\end{aligned}$$

$$\begin{aligned}
\Big([\![rel_var\texttt{(}term_1\texttt{,...,}term_n\texttt{) := }rel_expr]\!]_s(s)\Big)(r) &= s(r), && \text{if } r \neq rel_var \\
&\quad \Big\{ \big(v(term_1), \ldots, v(term_n)\big) \;\Big|\; v \in [\![rel_expr]\!]_e(s) \Big\} \\
&\cup \Big\{ t \in s(r) \mid \exists i : term_i \in \mathcal{U} \wedge term_i \neq t_i \Big\}, && \text{if } r = rel_var
\end{aligned}$$

with *attribute, attribute*$_1$*, attribute*$_2$*, attribute*$_3 \in \mathcal{X}$, *term*$_1$*, ..., term*$_n \in \mathcal{X} \cup \mathcal{U}$, and *rel_var* $\in \mathcal{R}$. The symbol $\overset{lfp}{=}$ denotes the least fixed point.

Fig. 2. RML semantics

References

AFC98. G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC 1998)*, pages 153–160. IEEE Computer Society, 1998.

BBS97. Ralf Behnke, Rudolf Berghammer, and Peter Schneider. Machine support of relational computations: The Kiel RELVIEW system. Technical Report 9711, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, 1997.

Bla04. Michael Blaha. A copper bullet for software quality improvement. *Computer*, 37(2):21–25, 2004.

BLM02. Rudolf Berghammer, Barbara Leoniuk, and Ulf Milanese. Implementation of relational algebra using binary decision diagrams. In H. de Swart, editor, *Proceedings of the 6th International Conference on Relational Methods in Computer Science (RelMICS 2001)*, LNCS 2561, pages 241–257, Berlin, 2002. Springer-Verlag.

BLQ+03. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 103–114. ACM, 2003.

BNL03. Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 216–225. IEEE Computer Society, 2003.

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

Bry92. Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

CGK98. Yih-Farn Chen, Emden R. Gansner, and Elftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9):682–694, 1998.

Ciu99. Oliver 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. IEEE Computer Society, 1999.

CM03. William F. Clocksin and Cristopher S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 5th edition, 2003.

FH00. Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 88–96. IEEE Computer Society, 2000.

FKvO98. Loe M. G. Feijs, René L. Krikhaar, and Rob C. van Ommering. A relational approach to support software architecture analysis. *Software: Practice & Experience*, 28(4):371–400, 1998.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.

Har91. John Hartman. Understanding natural programs using proper decomposition. In *Proceedings of the 13th International Conference on Software Engineering (ICSE 1991)*, pages 62–73, 1991.

HN90. Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, 1990.

Hol96. Richard C. Holt. Binary relational algebra applied to software architecture. Technical Report CSRI 345, University of Toronto, 1996.

Hol98. Richard 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. IEEE Computer Society, 1998.

IEE01. IEEE. *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) (IEEE Std 1003.1-2001)*, 2001.

KB98. Rick Kazman and Marcus Burth. Assessing architectural complexity. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR 1998)*, pages 104–112, 1998.

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

Kri01. Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 301–309. IEEE Computer Society, 2001.

KSRP99. Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick 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.

KW99. Bernt Kullbach and Andreas 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.

Mar97. Robert C. Martin. Engineering notebook: Stability. *C++ Report*, February 1997.

MNS01. Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

MS95. Alberto O. Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software – Concepts & Tools*, 16(4):170–182, 1995.

MWD99. Kim Mens, Roel Wuyts, and Theo D'Hondt. Declarative codifying software architectures using virtual software classifications. In *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems - Europe (TOOLS 1999)*, pages 33–45. IEEE Computer Society, 1999.

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

Qui94. Alex Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.

RW90. Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

SMB96. Forrest Shull, Walcélio L. Melo, and Victor 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.

SSC96. Mohlalefi Sefika, Aamod Sane, and Roy 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. IEEE Computer Society, 1996.

TA99. Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*, pages 230–238. IEEE Computer Society, 1999.

WHH02. Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Using graph patterns to extract scenarios. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 239–247. IEEE Computer Society, 2002.

Won98. Kenny Wong. *Rigi User's Manual, Version 5.4.4*, 1998. http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/.