Software architecture reconstruction
Krikhaar, R.

Citation for published version (APA):
Krikhaar, R. (1999). Software architecture reconstruction

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Appendix B

Abstraction Tools

B.1 Introduction

In this appendix we discuss some implementations of Relation Partition Algebra. A number of implementations have been created in a broad range of programming languages:

- functional language: Clean and Prolog
- scripting language: Perl and AWK
- database language: SQL
- imperative language: Pascal, C, C++, Basic, and Java

For each kind of language we will briefly discuss some issues of particular language.

One should not only think about the implementation of RPA operators, but also about the interface of these operators. We distinguish the following types of interfaces: API (application programmers interface) consisting of a set of functions (e.g. Java implementation), a command line (e.g. AWK implementation), a graphical calculator (e.g. SQL implementation) and a sophisticated graphical interface [Pet97]. These interfaces will not be discussed any further here.

B.2 RPA-Prolog

The first implementation of RPA was written in Prolog [SS86] using the SWI-Prolog interpreter [Wie96]. The key design decision for almost any
RPA implementation is the internal representation of sets and relations. For RPA-Prolog [Kri95] we have chosen the following data structure:

- A set is implemented as a list containing the elements.
- A relation is a compound term obj(SetX, SetY, Rel); with an invariant $SetX = \text{dom}(\text{Rel})$ and $SetY = \text{ran}(\text{Rel})$. Rel is a list of compound terms rel(Elem, ElemRel); where $ELEM \in SetX$ and $ElemRel$ is a list of elements with which $Elem$ has a relation. Each $Elem$ occurs only once in the $Rel$ list.
- The elements in the list $Rel$ and the elements in the list $ElemRel$ are ordered according to the order in $SetX$ and $SetY$, respectively.

The Prolog code of the rel_dom and rel_comp operators are listed below.

```prolog
/* Example of facts that represent a set and relation */

set(functions, [main,a,b,c,d]).
relation(calls, obj(functions, functions,
   [rel(main, [a,b]), rel(a, [b,c,d]),
    rel(b,[d]))]).

/*
   rel_dom(Relation, Set) <-
   Set is the domain of Relation
*/
rel_dom(obj(SetX, _, Rel), obj(SetX, Domain)) :-
   domain(Rel, Domain).
domain([rel(E,_)|R], [E|Dom]) :-
   domain(R, Dom).
domain([], []).

/*
   rel_comp(Relation1, Relation2, Relation) <-
   Relation is the composition Relation2 ; Relation1
*/
rel_comp(obj(NameX, NameY, Rel1), obj(NameY, NameZ, Rel2),
   obj(NameX, NameZ, Result)) :-
   set(NameY, SetY),
   set(NameZ, SetZ),
   comp(Rel1, SetY, SetZ, Rel2, Result).
comp([], -, -, -, []).
comp([rel(X,XList)|Rel1], SetY, SetZ,
   Rel2, [rel(X,XList)|Result]) :-
```


The input files for the AWK [AKW88] implementation consist of so-called RPA files. A set file contains a single element of the set at each line; in a relation file each line contains a tuple of two elements separated by white space. The lines in a multi-set file and a multi-relation file contain an
extra field to represent the weight.

We have implemented each RPA operator in a separate AWK script. The input of these scripts consists of files (including standard input) and the output is given on standard output. The *unix* pipe mechanism can be used to concatenate a number of operators, implementing for example an *excluded-lift* as depicted in Figure B.1:

```
rel_lift Relation1 Relation2 | rel_carX - Set3
```

A standard wrapper is used to parse the various arguments before the actual AWK script is called. This wrapper is responsible for checking the command line arguments, creating temporary files if needed (the *unix* way of referring to standard input `'-'` has been used to read from standard input), and some exception handling code. For clarity the wrapper code has been removed from the code below. Again, we give the source code for the `rel_dom` and `rel_comp` operators.

```bash
#!/bin/sh
# Calculates the domain of a relation
Usage="Call: rel_dom <rel>"
#
# <wrapper code> the variable $IN1 gets a value
#
awk 'seen[0]="" { print $1; seen[$1]=1; }'
  $IN1
```

```bash
#!/bin/sh
```

![Figure B.1: High Level Operations](image-url)
B.4 RPA-SQL

Another implementation is built on top of a database program. Any database that supports SQL would suffice, but we have used MS-Access [Boe96]. The various sets, relations, multi-sets and multi-relations are stored in separate tables. For relations, the columns in the table are named dom and ran, respectively. The table name refers to the relation’s name.

Query: rel_dom(<rel>):

```sql
SELECT DISTINCT <rel>.dom
FROM <rel>;
```

Query: rel_comp(<rel1>, <rel2>):

```sql
SELECT DISTINCT <rel1>.dom, <rel2>.ran
FROM <rel1> AS rel1 INNER JOIN <rel2> AS rel2
ON rel1.ran = rel2.dom
GROUP BY rel1.dom, rel2.ran;
```

Note that the rel_dom query refers to the relation name <rel>. In SQL it is however not possible to use such a construct. Therefore we have developed a Visual Basic program to instantiate such free variables in our SQL description. A new SQL statement is generated in which the actual values of these variables are filled out. After that, the generated SQL statement is applied to the data in the database.
Query: rel\_dom(calls):

```sql
SELECT DISTINCT calls.dom
FROM calls;
```

Query: rel\_comp(calls, calls):

```sql
SELECT DISTINCT calls.dom, calls.ran
FROM calls AS rel1 INNER JOIN calls AS rel2
ON rel1.ran = rel2.dom
GROUP BY rel1.dom, rel2.ran;
```

The stack-oriented RPA calculator shown in Figure B.2 has been built on top of this program. Sets and relations can be pushed on the stack, operations (represented by different buttons) are applied to the element(s) on the top of the stack.

### B.5 RPA-Java

In our Java [Web96] implementation of RPA we made much use of classes of standard packages. Various container classes of the `java.util` package were used; e.g., relations were represented in `HashSet`. We constructed an RPA package that contains the classes `Set`, `Relation`, `MultiSet`, and `MultiRelation`. Each class defines its own methods that perform related RPA operations. For example, the `rel\_dom` method of `Relation` calculates this object’s domain; the `rel\_comp` method of `Relation` calculates its composition with another `Relation` object. As already discussed in Section B.3, RPA files can be read and written by calling the provided IO methods.

The command line interface of the AWK implementation proved to be very handy in software architecture analysis. We have therefore implemented a similar interface on top of this Java implementation, consisting of a number of small programs, each calling a single method of the RPA package.

```java
public class Relation {

    // storage of tuples of Relation
    protected Hashtable tuples;

    /** dom( ) returns the domain of this Relation */
```
Figure B.2: RPA Calculator
public Set dom( ){
    Set s = new Set();
    Enumeration e = tuples.keys();
    while( e.hasMoreElements() ){
        s.insert( (String)( e.nextElement() ) );
    }
    return s;
}

/** comp( Relation r ) returns a Relation defined as r o 'this' */
public Relation comp( Relation r ){
    Relation res = new Relation();
    Enumeration e1 = r.tuples.keys();
    while( e1.hasMoreElements() ){
        String s1 = (String)( e1.nextElement() );
        Hashtable h1 = (Hashtable)( r.tuples.get( s1 ) );
        Enumeration e2 = h1.keys();
        while( e2.hasMoreElements() ){
            String s2 = (String)( e2.nextElement() );
            Hashtable h2 = (Hashtable)( s.tuples.get( s2 ) );
            if( h2 != null ){
                Enumeration e3 = h2.keys();
                int i1 = get( h1, s2 );
                while( e3.hasMoreElements() ){
                    String s3 = (String)( e3.nextElement() );
                    int i2 = get( h2, s3 );
                    res.insert( s1, s3, Integer( i1 * i2 ) );
                }
            }
        }
    }
    return res;
}

B.6 A Brief Comparison of RPA tools

The Prolog implementation should just be seen as a first experiment with the aim of becoming familiar with RPA.

The AWK implementation proved to be very suitable in daily practice. It
is easy to use (certainly for persons familiar with \textit{unix} concepts). The AWK scripts are interpreted and can be easily combined using other shell scripts. It is easy to incorporate these scripts, e.g. in a \textit{make} facility. A disadvantage may be the performance, the scripts may take some time in the case of large relations.

The SQL implementation performs poorly in the case of large relations. This holds especially for calculating a relation's transitive closure. The transitive closure is implemented as an extra program (Visual Basic) which iterates over a number of RPA operations (composition and union).

The advantage of Java is that it is platform-independent. It is also easy to integrate RPA in e.g. Java applets in a Web browser.