Reasoning about Rules in Deductive Databases

Marbod Hopfner[1] and Dietmar Seipel[2]

[1] University of Tübingen, Wilhelm–Schickard Institute for Computer Science
Sand 13, D – 72076 Tübingen, Germany
hopfner@informatik.uni-tuebingen.de

[2] University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

Abstract. We have implemented a PROLOG–tool VISUR/RAR for reasoning about rule–based systems, such as PROLOG–programs. RAR provides *retrieval* and *update operations* for a deductive database storing the investigated rules. The obtained results are graphically visualised or presented in tables using VISUR.
The tool VISUR/RAR has been used for analysing PROLOG–source code based on suitable *dependency graphs*. It can be applied for improving the design of rule–based systems, for computing certain *software metrics*, and for supporting *refactoring techniques*. Since it is mostly based on the implicational structure of rules, it can be adapted to various kinds of rules in expert systems.

Keywords. PROLOG, deductive databases, reasoning, retrieval

1 Introduction

For many programming languages, there exist powerful *Integrated Development Environments (IDEs)*, such as, e.g., IBM's ECLIPSE for JAVA [5] and Together for JAVA, C++, Visual Basic, etc. [11]. They support programmers in keeping track of a large project, and in correcting, completing and reusing source code. IDEs contain tools such as editors with *syntax highlighting* or tools for *graphical programming*; it might even be possible to generate graphical interfaces using drag–and–drop operations. Since the logic programming community is comparatively small, only few tools exist for comfortably programming and for analysing source code, cf., e.g., the IDE for XPCE–PROLOG [13] and the tool CIDER [3] for the functional–logic language CURRY.

We have developed the PROLOG–package VISUR/RAR, which provides some essential functionality of an IDE. It allows for the *visualisation* of rules (VISUR: Visualisation of Rules) together with the *inference* over rule structures (RAR: Reasoning about Rules). VISUR/RAR is a part of the library DISLOG [8], which is developed under XPCE/SWI–PROLOG. The functionality of DISLOG ranges from reasoning in disjunctive deductive databases to applications such as the management and visualisation of stock information.

Visur/Rar supports Prolog–developers in the *analysis* and the further development of their projects. Visur/Rar makes it easy for the user to become acquainted with the source code by visualising coherences between different source files of a project or by visualising rule calls. For example, it is possible to answer the following questions using Visur/Rar:

1. Which parts (files, modules) of a Prolog–project are needed in order to make a specific predicate work correctly ? This question is relevant, if one only wants to *export parts* of a project in order to pass on these parts to a third party; then it is sufficient to install only these parts in the new environment.
2. Which predicates call predicates from other modules, and which of them call the starting predicate again, i.e., are recursive ? Which predicates are never used (dead code) ? These questions are important, if one wants to *reorganise* the predicate locations and the source code.

The following Prolog–rules for reasoning about source code will be our running example throughout the paper. We will explain their meaning in Section 4, and we will organise them in three modules: `desc`, `necess`, and `cross`.

```
rar_predicate_to_descendants(Predicate, Descendants) :-
   transitive_closure( rar_predicate_to_descendant,
      Predicate, Descendants ).
rar_predicate_to_descendant(Predicate, Descendant) :-
   rar_search(rule, [head:Predicate], Rules),
   member(Rule, Rules), Descendants := Rule^body,
   member(Descendant, Descendants).

rar_predicate_to_necessary_files(Predicate, Files) :-
   rar_predicate_to_descendants(Predicate, Descendants),
   maplist( rar_predicate_to_files, Descendants, Fs ),
   append(Fs, Files).

rar_predicate_to_cross_calls(Predicate, Predicates) :-
   rar_predicate_to_descendants(Predicate, Descendants),
   findall( P,
      ( member(P, Descendants),
        not( in_same_module(Predicate, P) ) ),
      Predicates ).
in_same_module(M:_/_, M:_/_).
```

For example, the predicate `desc:rar_predicate_to_descendants/2` is relevant for `necess:rar_predicate_to_necessary_files/2`, whereas the predicate `cross:rar_predicate_to_cross_calls/2` is not relevant, and none of the predicates is recursive.

Using Rar it is possible to obtain further *analysis* that can be customised to the individual needs of a user, and to visualise the results graphically or in tables. We have already applied Visur/Rar to the source code of the system DisLog, which currently contains about 60.000 lines of Swi–Prolog–code in about 10.000 Prolog–rules. The goal is to support the application of software engineering and refactoring techniques and the further system development with Visur/Rar. Since Visur/Rar is mostly based on the implicational structure of rules, it could in principle be applied to various kinds of rules in expert systems.

The rest of the paper is organised as follows: In Section 2 we introduce some basic definitions and notations, which we use in further sections. In Section 3 we present the data structures and the basic operations of Rar, and we explain how the results can be shown with Visur. In Section 4 we investigate some typical problems and questions that might be asked about rules, and we show how we can solve these problems using Rar.

2 Basic Definitions and Notations

In Visur/Rar complex structured objects are stored using a special Prolog–term representation called *field–notation* [9], which will be sketched in Section 2.1. For visualising the call structure of rule–based systems the concept of *dependency graphs*, which is well–known from deductive databases [1], will be suitably extended in Section 2.2.

2.1 Complex Objects in Prolog

The *field–notation* of [9] will be explained as far as it is used in this paper. Simplified, the field–notation provides an easy way to access attribute/value–pairs in an *association list*. An association list looks like $[a_1 : v_1, \ldots, a_n : v_n]$, where a_i is called an *attribute* and v_i is called the associated *value*.

There exist several possibilities to *access* and *update* values in the field–notation. We will explain the possibilities in an example. Given an object `O` in field–notation, we use the call `X := O^A` to select the value `X` of an attribute `A`:

```
?- X := [a:1, b:2, c:3]^b.

X = 2
```

To change the values of attributes, the call `X := O*As` is used, where `As` is an association list specifying the new attribute/value–pairs and `X` is the updated object:

```
?- X := [a:1, b:2, c:3]*[^a:10, ^b:20].
X = [a:10, b:20, c:3]
```

Using the field–notation has got some advantages: The sequence of attribute/value–pairs is arbitrary. Values can be accessed by attributes rather than by argument positions. Null values can be omitted, and new values can be added at runtime.

2.2 Dependency Graphs with Meta–Predicates

DATALOG–programs can be analysed using diverse dependency graphs, e.g., the *rule/goal–graph* and the *goal–graph* [1]. Unfortunately, these graphs cannot handle practical applications of PROLOG properly, since they do not take into account meta–predicates.

Meta–predicates allow for higher–order programming. Well–known examples of meta–predicates are `call/1`, `findall/3`, `forall/2`, `maplist/3`, and `checklist/2`. For example, the evaluation of `maplist/3` in the following rule calls `rar_predicate_to_files(Descendant,F)` for all elements `Descendant` of the list `Descendants` and collects the results `F` in another list `Fs`:

```
rar_predicate_to_necessary_files(Predicate, Files) :-
   rar_predicate_to_descendants(Predicate, Descendants),
   maplist( rar_predicate_to_files, Descendants, Fs ),
   append(Fs, Files).
```

To treat *meta–predicates* in PROLOG–programs adequately, we will define extended dependency graphs, which can visualise meta–predicates in a way, that also shows the predicates called in the parameter list of a meta–predicate. Figure 1 shows the extended dependency graph of the rule; it was generated using the tool VISUR.

We use a *circle* for ordinary predicates; the name and the arity of the predicate are given below the symbol. For each rule, we use a *box*; the filename below the box gives the file in which the rule is defined. We use a *rhombus* as symbol for meta–predicates in order to distinguish them from ordinary predicates.

The Rule/Goal–Graph. Given a PROLOG–program P and a rule

$$r = A \leftarrow B_1 \wedge \ldots \wedge B_m \in P,$$

the concept of the rule/goal–graph $G_r^{rg} = \langle V_r^{rg}, E_r^{rg} \rangle$ of r is well–known from literature:

$$V_r^{rg} = \{ p_A, r \} \cup \{ p_{B_i} \mid 1 \leq i \leq m \},$$
$$E_r^{rg} = \{ \langle p_A, r \rangle \} \cup \{ \langle r, p_{B_i} \rangle \mid 1 \leq i \leq m \},$$

where p_X is the predicate name of an atom $X = p(t_1, \ldots, t_n)$, i.e. $p_X = p$. The rule/goal–graph of P is $G_P^{rg} = \bigcup_{r \in P} G_r^{rg}$.

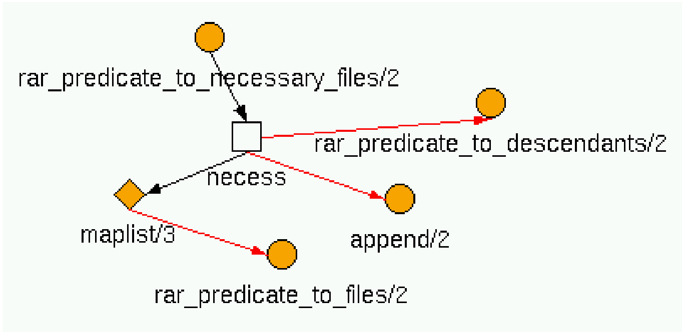

Fig. 1. Rule/Goal–Graph with Meta–Predicate in VISUR

Embedded Calls. We introduce the notation $B \prec A$ for expressing that an atom B is called within another atom A. E.g., for

$A =$ `maplist( rar_predicate_to_files, Descendants, Fs )` and
$B =$ `rar_predicate_to_files(Descendant, F)`

we get $B \prec A$. In our system it is possible to define the relation $\prec$ customised to individual user needs. E.g., if the predicate symbol p_B of the called atom B is constructed at runtime by appending some dynamic suffix to a static prefix, i.e. $p_B = p'_B \circ p''_B$, then we can specify that $B \prec A$ holds for all p_B with the prefix p'_B. In Section 3.2 we will see such predicates, where $p'_B = \mathsf{rar\_}$ and $p''_B \in \{\, \mathsf{search}, \mathsf{insert}, \mathsf{delete}, \mathsf{update} \,\}$.

The Extended Rule/Goal–Graph $G_P^{erg} = \langle\, V_P^{erg}, E_P^{erg} \,\rangle$ takes care of the fact that an atom B can be called within another atom A. G_P^{erg} is obtained by adding the following nodes and edges for the ground atoms A in the Herbrand base H_{BP}:

$$V_A^{\prec} = \{\, p_A \,\} \cup_{B \prec A} V_B^{\prec},$$
$$E_A^{\prec} = \cup_{B \prec A} (\, \{\, \langle\, p_A, p_B \,\rangle \,\} \cup E_B^{\prec} \,).$$

Further graphs such as the *goal–graph*, the *file dependency graph*, and the *module dependency graph* can be derived from the extended rule/goal–graph.

3 Data Structures and Basic Operations of VISUR/RAR

RAR provides the common database operations *search*, *insert*, *delete*, and *update* of rules in deductive databases. These operations are implemented on an *internal database*. Even though the RAR commands are collected in a module rar, all commands begin with the additional prefix rar_. We choose this naming, because it is not allowed to redefine built–in predicates of Swi–Prolog, such as delete/3. The results of Rar–operations can be visualised using the graphical user interface Visur.

Predicates defined in a module are not offhand accessible from other predicates that are not in the the same module. To access predicates from a module, the module name must be a prefix, separated with a colon from the predicate name. We use the naming convention Module:Predicate/Arity for predicates. If a predicate is not defined in a module, then it is automatically assigned to the global module user.

For defining a file as a module, the file must start with the declaration :- module(Module_Name, Exported_Predicates). Module_Name is the name of the module and Exported_Predicates is a list of predicates, defined and exported in the module Module_Name. The exported predicates can be accessed from every other predicate without using a prefix.

3.1 The Internal Database

The internal database of Rar consists of asserted facts in the module rar for the predicates rule, node, and edge. Each argument of an Rar–fact is an association list as shown in the following schema:

```
rule([id:I, file:F, head:H, body:B, code:C]).
node([id:I, symbol:Sy, size:Si, colour:C]).
edge([id:I, source:S, target:T, arrows:A, colour:C]).
```

E.g., in a fact with the predicate rule, I is a unique ID and F is the filename in which the rule is defined. H is the head predicate, B is a list of all body predicates, and C represents the rule in DisLog–notation.

Example 1. The rules in the Prolog–file *desc* are represented by the following Rar–facts:

```
rule([ id:3, file:'desc',
  head:(desc:rar_predicate_to_descendants/2),
  body:[extern:transitive_closure/3],
  code:
    [rar_predicate_to_descendants(X,Y)]-
      [transitive_closure(rar_predicate_to_descendant,X,Y)] ]).

rule([ id:4, file:'desc',
  head:(desc:rar_predicate_to_descendant/2),
  body:[rar:rar_search/3, extern:member/2,
    ddb_my_built_in:':='/2, extern:member/2],
```

```
code:
  [rar_predicate_to_descendant(X,Y)]-
    [rar_search(rule,[head:X],Z),
     member(W, Z),
     V := W^body,
     member(Y,V)] ]).
```

The RAR–facts for the predicates `node/1` and `edge/1` are used for storing dependency graphs, and they are used by the component VISUR.

3.2 Database Operations on the Internal Database

The RAR–commands operate on the internal RAR–database. The first parameter `Type` indicates which of the RAR–facts the operation should be applied to: `Type` can be `rule`, `node`, or `edge`.

Insertion into the Internal Database. Given an association list `As`, the call

```
rar_insert(+Type, +As).
```

asserts a fact `Type(As)` with the predicate symbol `Type` into the internal database of RAR. E.g., in order to insert a rule the command

```
rar_insert(rule, [id:I, file:F, head:H, body:B, code:C])
```

is issued. The unique ID `I` can be determined by the user or by the system. If the user determines this ID and another RAR–fact with the same ID already exists, then the insertion fails.

Search in the Internal Database. Given an association list `As`, the call

```
rar_search(+Type, +As, -Objects).
```

searches the internal database for the list `Objects` of all RAR–objects that match with all elements of the association list `As`. For example, if `Type = rule` and `As = [head:(M:P/A)]` it computes the *definition* of the predicate `M:P/A`, i.e., the list `Objects` of all RAR–rules that have `M:P/A` in their head. If `Type = rule` and `As = [head:(_:P/A), file:F]` it computes all rules in the file `F` with head `P` and arity `A`. If the searched predicate does not occur in a module, then the module name `user` is used.

The following rules collect the list `Files` of all filenames (possibly with duplicates), in which a (*multifile*) predicate `Predicate` is defined:

```
rar_predicate_to_files(Predicate, Files) :-
   rar_search(rule, [head:Predicate], Rules),
   maplist( rar:rar_filename,
      Rules, Files ).

rar_filename(Rule, File) :-
   File := Rule^file.
```

Deletion from the Internal Database. Internal RAR–facts are deleted by

```
rar_delete(+Type, +As, -Objects).
```

All internal RAR–facts which match with all attribute/value–pairs in the association list `As` will be deleted. A list `Objects` of the deleted RAR–facts will be returned.

Update of the Internal Database. Given two association lists `As_1` and `As_2`, the call

```
rar_update(+Type, +As_1, +As_2, -Changed_Objects)
```

updates all rules matching `As_1` by assigning the new values given in `As_2`. The updated rules are returned in the list `Changed_Rules`.

```
rar_update(Type, As_1, As_2, Changed_Objects) :-
   rar_delete(Type, As_1, Matched_Objects),
   maplist( rar_update_type(Type, As_2),
      Matched_Objects, Changed_Objects ).

rar_update_type(Type, As, Object, Changed_Object) :-
   Changed_Object := Object*As,
   rar_insert(Type, Changed_Object).
```

Example 2. Updating the filename of all predicates in the file desc to descendants is done as follows:

```
?-  rar_update(rule,
      [file:'desc'], [file:'descendants'], Rules).

Rules = [
   [ id:3, file:'descendants',
     head:(desc:rar_predicate_to_descendants/2),
     ... ],
   [ id:4, file:'descendants',
     head:(desc:rar_predicate_to_descendant/2),
     ... ] ].
```

3.3 Construction of the Rule/Goal–Graph

The component VISUR can be used for visualising the results of RAR–programs either graphically or in tables. It operates on the dependency graph, which has to be constructed using the RAR–operations of Section 3.2, where `Type` has the value `node` or `edge`.

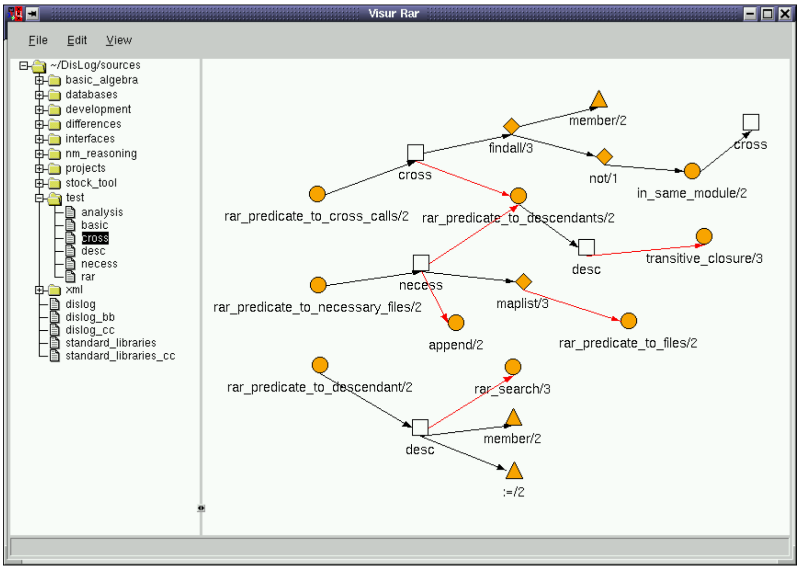

Fig. 2. Graphical User Interface of VISUR

Example 3. The rule/goal–graph of the three files given in Section 1 is shown in Figure 2, which has been generated using VISUR. The internal database representing this rule/goal–graph can be constructed as follows:

```
rar_built_graph(Rule) :-
   [Head, Body, H-Atoms] := Rule^[head, body, code],
   rar_insert(node,
      [ id:Head, symbol:circle, size:12, colour:yellow ]),
   rar_insert(node,
      [ id:R, symbol:box, size:12, colour:white ]),
   rar_insert(edge,
      [ id:E, source:Head, target:R,
        arrows:single, colour:black ]),
   checklist( rar:rar_insert_body(R),
      Body ),
```

```
    checklist( rar:rar_insert_embedded_predicates,
       Atoms ).

rar_insert_body(R, Predicate) :-
    rar_insert(node,
       [ id:Predicate, symbol:circle,
         size:12, colour:yellow ]),
    rar_insert(edge,
       [ id:E, source:R, target:Predicate,
         arrows:single, colour:black ]).

rar_insert_embedded_predicates(A1) :-
    forall( embedded_atom(A1, A2),
       ( functor(A1, P1, _), functor(A2, P2, _),
         rar_insert_body(P1, P2)
         rar_insert_embedded_predicates(A2) ) ).
```

For every rule the following steps are performed: Firstly, a node for the head predicate of a rule and a node for the rule itself are inserted, and the two nodes are connected by an edge. Then, a node for each body predicate in the rule is inserted and it is connected with the node for the rule. Finally, the predicates that are called with meta–predicates are treated by `rar_insert_embedded_predicates`. The predicate `embedded_atom` can be any user–defined predicate expressing which calls are considered as embedded calls in the application. The embedded call from `transitive_closure/3` to `rar_predicate_to_descendant/2` cannot be seen in Figure 2, since – when the screenshot was made – the configuration of the system did not contain `transitive_closure/3` as a meta–predicate.

4 Reasoning about Rules in Rar

Some interesting questions about source code are, "Is the arrangement of the predicates in the source code files suitable ?", "Which predicates are needed so that a predicate works correctly ?", or "Is there unused (dead) code in the source code ?". In this section, we will show by some case studies how the basic Rar–operations introduced in Section 3 can be used for answering questions of this type. The questions will be answered by using Prolog–rules; interestingly, we can take these Prolog–rules also as examples which we analyse.

4.1 Computing Necessary Predicates

`rar_predicate_to_descendants/2` returns the list `Descendants` of all predicates that are directly or indirectly called by a given predicate `Predicate`; these predicates are *relevant* for `Predicate` to work correctly:

```
rar_predicate_to_descendants(Predicate, Descendants) :-
   transitive_closure( rar_predicate_to_descendant,
      Predicate, Descendants ).

rar_predicate_to_descendant(Predicate, Descendant) :-
   rar_search(rule, [head:Predicate], Rules),
   member(Rule, Rules),
   Descendants := Rule^body,
   member(Descendant, Descendants).
```

The predicate `transitive_closure/3` computes the list `Descendants` of all predicates that are transitively called by `Predicate`. `Predicate` is *recursive*, if it occurs in `Descendants`. E.g., the call

```
?- rar_predicate_to_descendants(
      desc:rar_predicate_to_descendant/2, Descendants).

Descendants = [
   rar:rar_search/3, extern:member/2, ddb_my_built_in:':='/2 ]
```

shows that the predicate `desc:rar_predicate_to_descendant/2` is non–recursive. The computed predicates in `Descendants` are necessary for it, whereas the predicate `rar:rar_predicate_to_cross_calls/3` is irrelevant.

If the head predicate of a rule is not (transitively) called from any of the top–level predicates of a system, then the rule most likely is *dead code*. For real code, we also have to look at the arguments of meta–predicates, and we have to consider new rules asserted at runtime which might be calling the supposedly dead code.

4.2 Computing Necessary Files

The predicate `rar_predicate_to_necessary_files/2` calculates all necessary files for a predicate in order to work correctly:

```
rar_predicate_to_necessary_files(Predicate, Files) :-
   rar_predicate_to_descendants(Predicate, Descendants),
   maplist( rar_predicate_to_files,
      Descendants, Fs ),
   append(Fs, Files).
```

Since `Fs` is a list of lists, it is *flattened* by appending these lists to a single list `Files`.

4.3 Searching for Calls Across Modules

The predicate `rar_predicate_to_cross_calls/2` determines all descendants which are not in the same module as the calling predicate:

```
rar_predicate_to_cross_calls(Predicate, Predicates) :-
   rar_predicate_to_descendants(Predicate, Descendants),
   findall( P,
      ( member(P, Descendants),
        not( in_same_module(Predicate, P) ) ),
      Predicates ).

in_same_module(M:_/_, M:_/_).
```

All rules which have predicates from another module in the body are relevant. For the list of all head predicates we call `rar_predicate_to_cross_calls/2`, and we collect the predicates with a non–empty result list.

Example 4. The module dependency graph of the PROLOG–rules of Section 4 is shown in Figure 3. We assume that the PROLOG–rules are arranged in the three modules `desc`, `necess`, and `cross`. Again we can see the hierarchical structure of the PROLOG–program. The virtual module `extern` stands for example for the predicate `member/2`:

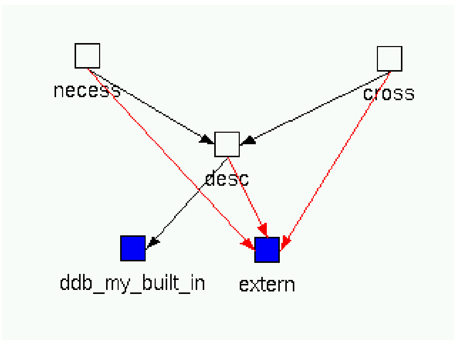

Fig. 3. Module Dependency Graph

The complete rule/goal–graph of these rules is given in Figure 2. The embedded call from `transitive_closure/3` to `rar_predicate_to_descendant/2` cannot be seen, since – when the screenshot was made – the configuration of the system did not contain `transitive_closure/3` as a meta–predicate. The meta–predicates can be defined by the user of VISUR/RAR at runtime. In the screenshot `findall/3`, `maplist/3`, and `not/1` can be seen as meta–predicates.

5 Conclusions

VISUR/RAR can be used for analysing source code and for visualising the results. By using an extended definition of dependency graphs it became possible to treat meta–predicates adequately. We have motivated our concepts via some examples, and we have explained by some case studies how user–defined predicates for various purposes can be implemented using RAR.

VISUR/RAR is a very flexible tool that allows for analysing and visualising various types of complex structures, such as, e.g., PROLOG–rules. In [4] it has been shown how even *ER–diagrams* can be visualised by VISUR.

In general, we have to deal with some more problems, such as rules that are asserted at runtime. In the future, we will gradually extend VISUR/RAR with additional features. E.g., it is conceivable to integrate a possibility to move predicates from one location in a file to another location in another file (or the same file) using drag–and–drop operations; heuristic rules in RAR might be used for making sure that the move does not change the meaning of the program.

We also intend to implement sophisticated methods for *program analysis* from *software engineering* [2, 6, 7], and we want to integrate *refactoring techniques* for PROLOG– and for JAVA–code, which have been developed in [10]. We also consider applying VISUR/RAR to other programming languages; e.g., we plan to adapt our methods to an XML–representation of JAVA–programs.

References

1. *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.
2. *M. Fowler:* Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.
3. *M. Hanus, J. Koj:* CIDER: An Integrated Development Environment for Curry, Proc. Workshop on Functional and Logic Programming WFLP'2001.
4. *M. Hopfner:* Eine graphische Oberfläche zur Verwaltung und zum Retrieval von Regeln in deduktiven Datenbanken, Diploma Thesis, University of Würzburg, 2002.
5. The Integrated Development Environment ECLIPSE, http://www.eclipse.org/
6. *J. Seemann, J. Wolff von Gudenberg:* Pattern–Based Design Recovery of JAVA Software, Proc. Intl. Symposium on the Foundations of Software Engineering 1998.
7. *J. Seemann, J. Wolff von Gudenberg:* Software–Entwurf mit UML, Springer, 2000.
8. *D. Seipel:* DISLOG – A Disjunctive Deductive Database Prototype, Proc. 12th Workshop on Logic Programming WLP'1997.
9. *D. Seipel:* Processing XML–Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP'2002.
10. *R. Seyerlein:* Refactoring in deduktiven Datenbanken am Beispiel des Informationssystems Qualimed, Diploma Thesis, University of Würzburg, 2001.
11. *TogetherSoft Corporation:* Together Control Center, http://www.togethersoft.com
12. *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/
13. *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG http://www.swi-prolog.org/