Comprehending and Visualising Software based on XML–Representations and Call Graphs

Marbod Hopfner[1] Dietmar Seipel[2]
Jürgen Wolff von Gudenberg[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,wolff}@informatik.uni-wuerzburg.de

Abstract

We have implemented a PROLOG*–tool* VISUR/RAR *for reasoning about various types of source code, such as* PROLOG*–rules or* JAVA*–programs.* RAR *provides* retrieval *and* update operations *for a deductive database storing* XML*–representations of the investigated code. The obtained results are visualised using graphs or tables in* VISUR.

The deductive database contains rules for analysing PROLOG*–code based on suitable* dependency graphs *and rules for recovering the design of* JAVA*–software using a query language* FNQUERY *that is based on path expressions. It can be applied for improving the design of rule–based systems, for computing certain* software metrics*, and for supporting* refactoring techniques.

Keywords. comprehension and visualisation, JAVA, PROLOG, reasoning, retrieval, XML

1. Introduction

For many programming languages, there exist powerful *Integrated Development Environments (IDEs)*, such as IBM's ECLIPSE for JAVA [8], and Together for JAVA, C++, Visual Basic, etc. [13]. 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 [15] and the tool CIDER [5] 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 [10], 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.

The system VISUR/RAR supports 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 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.

3. Which relationships exist between classes and methods in JAVA–source code ?

Using RAR it is possible to *analyse* source code 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. In principle, VISUR/RAR could be applied to various kinds of rules in expert systems.

We have also applied RAR for analysing JAVA–code. We use an XML–notation called JAML for representing the JAVA–code and a library called FNQUERY for processing the JAML–documents. In particular, we show a collection of PROLOG–rules for design recovery. Interestingly, we can visualise these rules using VISUR, and thus we use them as illustrating examples throughout the paper.

The rest of the paper is organised as follows: In Section 2 we introduce XML–representations for JAVA and PROLOG, and we show how this XML–code can be managed in PROLOG. In Section 3 we briefly describe the tool VISUR for visualising dependency graphs. In Section 4 we investigate some typical problems and questions that might be asked about JAVA–code or PROLOG–rules, and we show how we can solve these problems using RAR.

2. Representation of Source Code

In VISUR/RAR complex structured objects – i.e., JAVA–programs and PROLOG–programs – are conceptionally treated in XML–notation, cf. Section 2.1 and 2.2, which itself is physically stored in a PROLOG–term representation called *field–notation* [11], cf. Section 2.3.

2.1. JAVA–Code in JAML

The following JAVA–class implements the well–known sorting algorithm *merge sort* on a "globally defined" array. The method for merging two sorted sub–arrays is not shown here.

```
class MergeSort {
   ...
   public void sort (int l, int r) {
      if (l<r) {
         int m = (l+r)/2;
         sort(l,m);
         sort(m+1,r);
         merge(l,m,r);
      }
   }
}
```

In [3] an XML–language JAML is introduced for representing the parse trees of JAVA–code. For example, in the following we show the JAML–representation for a small portion of the JAVA–code from above, namely the method invocation `sort(l,m)`:

```
<method-invocation
   name="sort" qualifier=""
   argument-count="2"
   argument-0="l" argument-1="m"
   signature= "sort(int, int)"
   return-type-ref="void">
   <literal-expression
      literal="sort" type-ref="">
      <identifier>sort</identifier>
   </literal-expression>
   <symbol kind="left-parenthesis">
      ( </symbol>
   <literal-expression
      literal="l" type-ref="int">
      <identifier>l</identifier>
   </literal-expression>
   <symbol kind="comma">
      , </symbol>
   <literal-expression
      literal="m" type-ref="int">
      <identifier>m</identifier>
   </literal-expression>
   <symbol kind="right-parenthesis">
      ) </symbol>
</method-invocation>
```

2.2. PROLOG–Code in XML

PROLOG–rules can be represented in XML according to the following DTD:

```
<!ELEMENT rule (head, body)>
<!ATTLIST rule
   id CDATA #required
   file CDATA #required>
<!ELEMENT head atom>
<!ELEMENT body atom*>
<!ELEMENT atom arg*>
<!ATTLIST atom
```

```
   pred CDATA #required>
<!ELEMENT arg CDATA #required>
```

For example, the following rule will be used in Section 4.2 within a program for extracting design information from JAVA–programs. It expresses that a class `C1` creates another class `C2`, if it owns a method `M1` that creates `C2`:

```
creates_cc(C1,C2) :-
   owns_cm(C1,M1),
   creates_mc(M1,C2).
```

This rule can be represented in XML as follows:

```
<rule id="2" file="analysis">
   <head>
      <atom pred="creates_cc">
         <arg>C1</arg>
         <arg>C2</arg>
      </atom>
   </head>
   <body>
      <atom pred="owns_cm">
         <arg>C1</arg>
         <arg>M1</arg>
      </atom>
      <atom pred="creates_mc">
         <arg>M1</arg>
         <arg>C2</arg>
      </atom>
   </body>
</rule>
```

2.3. Complex Objects in PROLOG

In PROLOG, a complex object can be represented as an *association list* $[a_1 : v_1, \ldots, a_n : v_n]$, where a_i is an *attribute* and v_i is the associated *value*. In the library FNQUERY [11] this formalism has been extended to the *field–notation* for XML–documents: an XML–object

$$\langle T\ a_1 = "v_1" \ldots a_n = "v_n" \rangle \ldots \langle / T \rangle$$

with the tag "T" can be represented as a PROLOG–term $T : As : C$, where $As = [a_1 : 'v_1', \ldots, a_n : 'v_n']$ is an association list for the attribute/value–pairs and C represents the contents (i.e., the sub–elements). E.g., for the rule of Section 2.2 we get:

```
rule:[id:2, file:analysis]:[
   head:[ ]:[
      atom:[pred:creates_cc]:[
         arg:[ ]:['C1'],
         arg:[ ]:['C2'] ] ],
   body:[ ]:[
      atom:[pred:owns_cm]:[
         arg:[ ]:['C1'],
         arg:[ ]:['M1'] ],
      atom:[pred:creates_mc]:[
         arg:[ ]:['M1'],
         arg:[ ]:['C2'] ] ] ]
```

There exist several possibilities to *access* and *update* objects `O` in field–notation. We use the call `X := O^A` to select the `A`–sub–element of `O`, and we use `Y := O@A` to select the value `Y` of the attribute `A`:

```
?- O = atom:[pred:owns_cm]:[
          arg:[ ]:['C1'],
          arg:[ ]:['M1'] ],
   X := O^arg, Y := O@pred.

X = arg:[ ]:['C1'],
Y = owns_cm
```

On *backtracking* all sub–elements with the given tag `arg` could be obtained.

To change the values of attributes or sub–elements, the call `X := O*As` is used, where `As` specifies the new attribute/value–pairs in the updated object `X`:

```
?- O = atom:[pred:owns_cm]:[
          arg:[ ]:['C1'],
          arg:[ ]:['M1'] ],
   X := O*[@pred:p, ^arg:['C3']].

X = atom:[pred:p]:[
       arg:[ ]:['C3'],
       arg:[ ]:['C3'] ],
```

Using the field–notation has got several 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.

The library FNQUERY also contains additional, more *advanced methods*, such as the selection/deletion of all elements/attributes of a certain pattern, the transformation of sub–components according to substitution rules in the style of XSLT, and the manipulation of path or tree expressions.

3. Visualisation of Rules in VISUR

For visualising the call structure of rule–based systems the concept of *dependency graphs*, which is well–known

from deductive databases [2], will be used. DATALOG–programs can be analysed using diverse dependency graphs, e.g., the *rule/goal–graph* and the *goal–graph* [2].

All screenshots of dependency graphs that are shown in this paper have been obtained using our system 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.

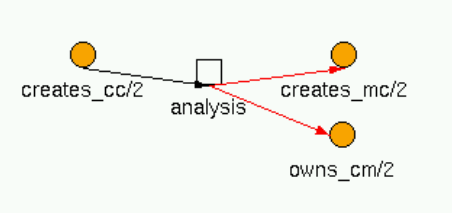

Figure 1. Rule/Goal–Graph in VISUR

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:

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

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 $G_P^{rg} = \bigcup_{r \in P} G_r^{rg}$.

Meta–Predicates in PROLOG 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 are `call/1`, `findall/3`, `forall/2`, `maplist/3`, and `checklist/2`. E.g., the evaluation of `maplist/3` within the following rule calls `rar_predicate_to_files/2` for all elements of the list `Descendants` and collects the results in the 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 also show the predicates called in the parameter list of a meta–predicate. Figure 2 shows the extended dependency graph of the rule generated by VISUR. We use a *rhombus* as the symbol for meta–predicates in order to distinguish them from ordinary predicates. In the rule/goal–graph there would be no edge from `maplist/3` to `rar_predicate_to_files`.

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

```
A = maplist(
       rar_predicate_to_files,
       Descendants, Fs ),
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 4 we will see such predicates, where $p'_B =$ rar_ and $p''_B \in \{$ search, insert, delete, update $\}$.

The Extended Rule/Goal–Graph. 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 HB_P:

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

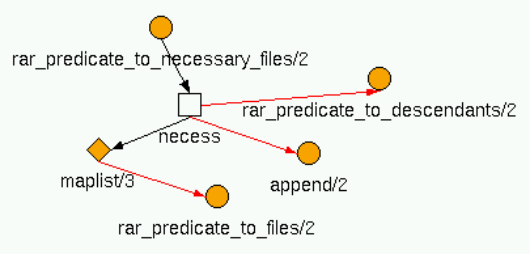

Figure 2. Extended Rule/Goal–Graph in VISUR

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.

4. Reasoning about Rules in RAR

In Section 4.1 we present the basic data structures and operations of the tool RAR. In the following subsections we apply RAR to JAVA–source code and to PROLOG–source code, respectively.

In Section 4.2 we give PROLOG–rules for extracting design information from JAVA–programs. These rules derive some useful relationships between methods and classes. The goal–graph of these rules is given in Figure 3.

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 the Sections 4.3 to 4.5, we will demonstrate by some case studies how the basic RAR–operations can be used for answering questions of this type; in Figure 7 the rule/goal–graph of the used rules is given. In our examples we apply the rules in the Sections 4.3 to 4.5 to the PROLOG–rules of Section 4.2.

4.1. Data Structures and Basic Operations

The *internal database* of RAR consists of asserted facts in the module `rar` for the predicates `rule`, `node`, and `edge`. The RAR–facts for the predicate `rule`, which are mainly used by RAR, contain the field notation for the XML–representation of PROLOG–rules. 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.

RAR provides the basic database operations *search*, *insert*, *delete*, and *update* of rules in deductive databases.

```
rar_insert(
   +Type, +As).
rar_search(
   +Type, +As, -Objects).
rar_delete(
   +Type, +As, -Objects).
rar_update(
   +Type, +As_1, +As_2,
   -Changed_Objects)
```

`rar_search` searches the internal database for the list `Objects` of all RAR–objects that match with all elements of the association list `As`, where `Type` can be `rule`, `node`, or `edge`. For example, the following call returns the list `Rules` of all rules that have a given predicate `Predicate` in their head:

```
rar_search(rule,
   [head:Predicate], Rules)
```

These operations are implemented on the 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.

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`.

4.2. Extracting Design Information

In the following we will recover design information from JAVA–source code in JAML–representation in two steps; the rules for analysing the code are separated into the two files `basic` and `analysis`. The goal–graph of the rules has been visualised using VISUR in Figure 3.

The first file, `basic`, defines (among others) some basic relations between methods and classes using path expressions in FNQUERY. For example, `references_cc/3` describes that the class `C1` has an attribute of the type `C2`. This holds, if at arbitrary depth in the JAML–representation `Code` of the source code there exists a `class-definition`–element `U` which itself contains a `field-declaration` `V` (at arbitrary depth) with an `identifier`–subelement. In that case `C1` is taken to be the `name`–attribute of `U` and `C2` is the `type`–attribute of `V`.

```
references_cc(Code,C1,C2) :-
   U := Code^_^class-definition,
   V := U^_^field-declaration,
   C1 := U@name,
   C2 := V@type,
   _ := V^identifier.

calls_mm(Code,M1,M2) :-
   U := Code^_^method-declaration,
   V := U^_^method-invocation,
   M1 := U@signature,
   M2 := V@signature.

owns_cm(Code,C,M) :-
   U := Code^_^class-definition,
   V := U^_^method-declaration,
   C := U@name,
   M := V@signature.
```

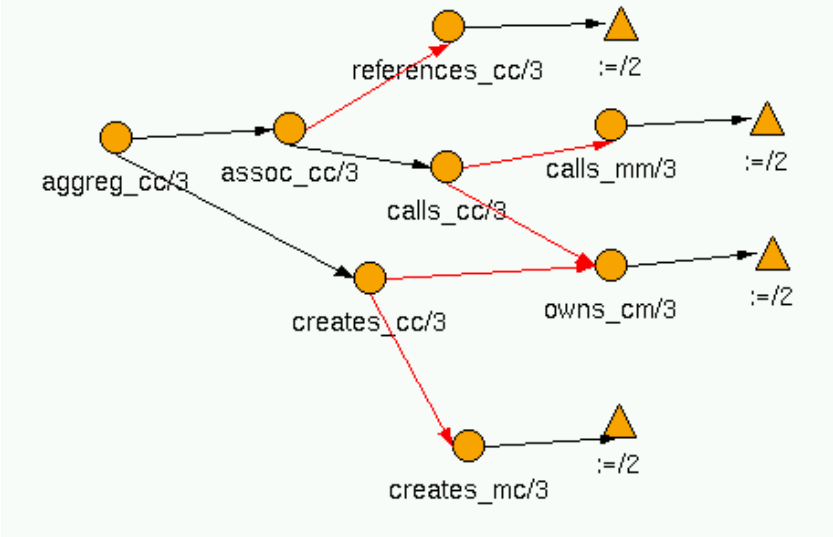

Figure 3. Goal–Graph in VISUR

```
creates_mc(Code,M,C) :-
   U := Code^_^method-declaration,
   V := U^_^instance-creation,
   M := U@signature,
   C := V@class-type.
```

The second file, `analysis`, contains another layer of rules, that are based on the basic predicates. These rules allow for a more complex analysis of the JAVA–source code. They have been proposed in [9] for the pattern–based design recovery of JAVA software. For example, *aggregations* and *associations* provide a higher–level view of the original *design* in contrast to the implementational view of the source code:

```
calls_cc(Code,C1,C2) :-
   calls_mm(Code,M1,M2),
   owns_cm(Code,C1,M1),
   owns_cm(Code,C2,M2).

creates_cc(Code,C1,C2) :-
   owns_cm(Code,C1,M1),
   creates_mc(Code,M1,C2).

assoc_cc(Code,C1,C2) :-
   calls_cc(Code,C1,C2),
   references_cc(Code,C1,C2).

aggreg_cc(Code,C1,C2) :-
   assoc_cc(Code,C1,C2),
   creates_cc(Code,C1,C2).
```

4.3. 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(
      analysis:calls_cc/3,
      Descendants).

Descendants = [
   basic:calls_mm/3,
```

```
    basic:owns_cm/3,
    ddb_my_built_in:(:=)/2 ]
```

derives the predicates that are necessary for `analysis:calls_cc/3`. We can see that, for example, `analysis:aggreg_cc/3` and `analysis:creates_cc/3` are irrelevant, and that `analysis:calls_cc/3` is non–recursive.

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*. In general, 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.

4.4. Computing Necessary Files

The following predicate 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`. For example, we get

```
?- rar_predicate_to_necessary_files(
      analysis:calls_cc/3, Files).

Descendants = [analysis, basic]
```

since these are the files that contain predicates that are descendants of `analysis:calls_cc/3`.

4.5. Searching for Calls Across Modules

The following predicate 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),
        \+ 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 the predicate `rar_predicate_to_cross_calls/2`, and we collect the predicates with a non–empty result list. For example, we get

```
?- rar_predicate_to_cross_calls(
      analysis:calls_cc/3, Ps).
Ps = [
   basic:calls_mm/3,
   basic:owns_cm/3 ]

?- rar_predicate_to_cross_calls(
      analysis:owns_cm/3, Ps).
Ps = [ ddb_my_built_in:(:=)/2 ]
```

The *module dependency graph* shows the calls across modules. Figure 4 shows this graph for the PROLOG–rules of Section 4.2, which are arranged in two modules `analysis` and `basic`. The virtual module `ddb_my_built_in` stands for the definition of the binary predicate "`:=`":

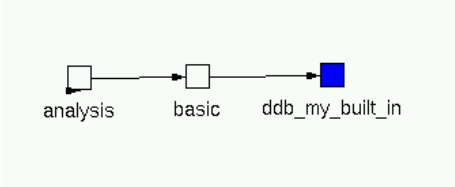

Figure 4. Module Dependency Graph

This module dependency graph shows the hierarchical structure of the investigated PROLOG–program, since there are no cycles. If a module dependency graph contains *cycles*, then we can try to improve the design of the source code by moving certain rules from one module to an other module until (most of) the cycles disappear.

The *calls across modules* of the rules in Section 4.2 are counted in Table 5, which was also generated by VISUR: there are 6 calls from `analysis` to `basic` and 17 calls from `basic` to `ddb_my_built_in`, namely the calls to the binary predicate "`:=`":

The module dependency graph of the PROLOG–rules of Sections 4.3 to 4.5 is shown in Figure 6. We assume that the PROLOG–rules are arranged in the three modules `desc`, `necess`, and `cross`. Again we can see the hierarchi-

module_1	module_2	calls
analysis	basic	6
analysis	ddb_my_built_in	0
basic	analysis	0
basic	ddb_my_built_in	17
ddb_my_built_in	analysis	0
ddb_my_built_in	basic	0

Figure 5. Calls accross Modules

cal structure of the PROLOG–program. The virtual module `extern` stands for example for the predicate `member/2`:

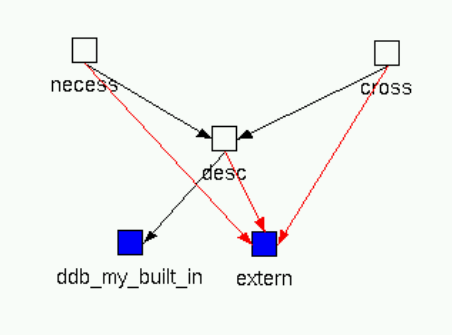

Figure 6. Module Dependency Graph

The complete rule/goal–graph of these rules is given in Figure 7. The embedded call from the predicate `transitive_closure/3` to the predicate `rar_predicate_to_descendant` 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`, `maplist`, and `not` can be seen as meta–predicates:

5. Conclusions

VISUR/RAR has been fully implemented in XPCE–PROLOG [14, 15]. It 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 including PROLOG–rules and JAVA–statements. In [7] it has been shown how even *ER–diagrams* can be mapped to suitable PROLOG–rules, such that the dependency graph of these rules visualises the ER–diagram.

The integration of XML–processing, visualisation, and reasoning in the logic programming environment XPCE–PROLOG has created a powerful and flexible tool. The library FNQUERY, which we are using from accessing the components of the XML–representations of the source code, has also been used in other projects. For example, in [6] it is shown how mathematical knowledge in MATHML can be managed nicely using FNQUERY.

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* [4, 9], and we want to integrate *refactoring techniques* for PROLOG– and for JAVA–code, which have been developed in [12].

References

[1] *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.

[2] *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.

[3] *G. Fischer, J. Wolff von Gudenberg:* JAML – An XML–Representation of JAVA–Source Code, submitted to IWPC'2003.

[4] *M. Fowler:* Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.

[5] *M. Hanus, J. Koj:* CIDER: An Integrated Development Environment for Curry, Proc. Workshop on Functional and Logic Programming WFLP'2001.

[6] *B. Heumesser, D. Seipel, U. Güntzer:* Flexible Processing of XML–Based Mathematical Knowledge in a PROLOG–Environment, Proc. Intl. Conf. on Mathematical Knowledge Management MKM'2003, Springer LNCS (to appear).

[7] *M. Hopfner:* Eine graphische Oberfläche zur Verwaltung und zum Retrieval von Regeln in deduk-

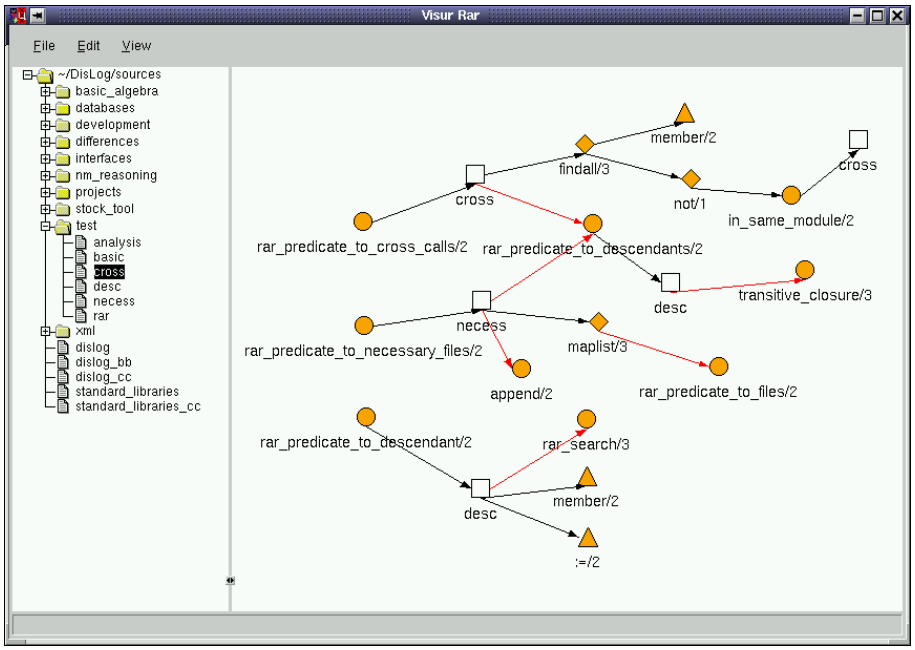

Figure 7. Graphical User Interface of VISUR

tiven Datenbanken, Diploma Thesis, University of Würzburg, 2002.

[8] *IBM:* The Integrated Development Environment ECLIPSE, http://www.eclipse.org/

[9] *J. Seemann, J. Wolff von Gudenberg:* Pattern–Based Design Recovery of JAVA Software, Proc. Intl. Symposium on the Foundations of Software Engineering 1998.

[10] *D. Seipel:* DISLOG – A Disjunctive Deductive Database Prototype, Proc. 12th Workshop on Logic Programming WLP'1997.

[11] *D. Seipel:* Processing XML–Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP'2002.

[12] *R. Seyerlein:* Refactoring in deduktiven Datenbanken am Beispiel des Informationssystems Qualimed, Diploma Thesis, University of Würzburg, 2001.

[13] *TogetherSoft Corporation:* Together Control Center, http://www.togethersoft.com

[14] *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/

[15] *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG http://www.swi-prolog.org/

[16] XML Query Requirements, Working Draft of the World Wide Web Consortium (W3C), http://www.w3.org/TR/2001/WD-xmlquery-req-20010215.