Analyzing and Visualizing PROLOG Programs based on XML Representations

Dietmar Seipel[1], Marbod Hopfner[2],
Bernd Heumesser[2]

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

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

Abstract

We have developed a PROLOG package VISUR/RAR for reasoning about various types of source code, such as PROLOG rules, JAVA programs, and XSLT stylesheets. RAR provides techniques for *analyzing* and *improving* the *design* of PROLOG programs, and it allows for implementing *software engineering metrics* and *refactoring techniques* based on XML representations of the investigated code. The obtained results are visualized by graphs and tables using the component VISUR.

VISUR/RAR can significantly improve the development cycle of logic programming applications, and it facilitates the implementation of techniques for syntactically analyzing and visualizing source code. In this paper we have investigated the dependency structure between the different rules and the hierarchical structure of PROLOG software systems, as well as the internal structure of individual predicate definitions.

For obtaining efficiency and for representing complex deduction tasks we have used techniques from deductive database and non–monotonic reasoning.

Keywords. comprehension, refactoring, reasoning, visualization, PROLOG, XML

1 Introduction

For many programming languages, powerful *integrated development environments (IDEs)* have been developed, such as IBM's Eclipse for JAVA [12], and Together for JAVA, C++, Visual Basic, etc. They contain tools such as editors with *syntax highlighting*, *tracing* and *debugging* and tools for *graphical programming*. Advanced IDEs support programmers in managing large projects, e.g. by facilitating the tasks of correcting, completing and reusing source code. In the logic programming community, so far only few tools exist for comfortably programming and for analyzing source code, cf., e.g., the IDEs for XPCE–PROLOG [20] and for Visual PROLOG, and the tool Cider [7] for the functional–logic language Curry.

The package VISUR/RAR [9] provides some essential functionality of an IDE for PROLOG. It allows for the *visualization* of rules (VISUR: Visualization of Rules) together with the *inference* over rule structures (RAR: Reasoning about Rules). VISUR/RAR is a part of the toolkit DISLOG, which is developed under XPCE/SWI–PROLOG [20]. The functionality of DISLOG ranges from (non–monotonic) reasoning in disjunctive deductive databases to applications such as the management and visualization of stock information.

The goal of the system VISUR/RAR is to support the application of *software engineering* and *refactoring* techniques, and the further system development. VISUR/RAR facilitates program comprehension and review, design improvement by refactoring, the extraction of subsystems, and the computation of software metrics (such as, e.g., the degree of abstraction). It helps programmers in becoming acquainted with source code by visualizing dependencies between different predicates or source files of a project. It is possible to analyse source code customized to the individual needs of a user, and to visualize the results graphically or in tables.

VISUR/RAR can be applied to various kinds of rule–based systems, including expert systems, diagnostic systems, XSLT stylesheets, etc. In this paper we have applied VISUR/RAR to the source code of the system DISLOG, which currently contains about 80.000 lines of code in SWI–PROLOG in about 10.000 PROLOG rules. In previous papers [9, 10] we have shown how also JAVA source code can be analysed using VISUR/RAR. For gaining sufficient performance on large programs such as DISLOG we use techniques from the field of deductive databases.

The rest of the paper is organized as follows: In Section 2 we introduce our PROLOG library for managing XML documents. In Section 3 we briefly describe the graph visualization tool VISUR. In Sections 4 and 5 we investigate some typical problems and questions that might be asked about the global and the local structure of PROLOG rules, respectively, and we show how we can solve these problems using RAR.

2 Representation of Source Code

In VISUR/RAR complex structured objects, such as JAVA programs, PROLOG programs, and XSLT stylesheets, are conceptually treated in two different XML notations; for handling these XML data we use the library FNQUERY, which is part of the DISLOG unit `xml` [17].

Firstly, RAR uses a notation which is similar to RuleML [19] for representing PROLOG defintions and rules; our DTD differs slightly from RuleML, and moreover we only use some of the elements and attributes mentioned in RuleML. Secondly, VISUR transforms our XML representation of rules and our reportings into the *Graph eXchange Language* GXL [11]; we added some additional attributes to the GXL notation for configuring the graph display.

2.1 PROLOG Programs in XML

The following PROLOG predicate `tc/2` computes the transitive closure of the relation `arc/2`:

```
tc(U1, U2) :-
   arc(U1, U3), tc(U3, U2).
tc(U1, U2) :-
   arc(U1, U2).
```

We are representing PROLOG programs in XML according to a DTD (see appendix), which is suitable for handling disjunctive logic programs with arbitrarily many head atoms. The arguments of an atom are either terms or variables; constants are represented as terms without subterms, where the constant is stored in the attribute `functor`. E.g., the PROLOG rules for the predicate `tc/2` are represented as follows:

```
<definition predicate="(user:tc)/2">
   <rule file="transitive_closure">
      <head>
         <atom predicate="(user:tc)/2">
            <var name="U1"/> <var name="U2"/> </atom>
      </head>
      <body>
         <atom predicate="(user:arc)/2"> ... </atom>
         <atom predicate="(user:tc)/2"> ... </atom>
      </body>
   </rule>
   ...
</definition>
```

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`.

2.2 Complex Objects 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*; this representation is well–known from the field of artificial intelligence. Using the field notation has got several advantages compared to ordinary PROLOG facts "$\mathsf{object}(v_1, \ldots, v_n)$". 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.

In the PROLOG library FNQUERY this formalism has been extended to the *field notation* for XML documents: an XML element

$$\langle \mathsf{Tag}\ a_1 = "v_1" \ldots a_n = "v_n" \rangle\ \mathsf{Contents}\ \langle/\mathsf{Tag}\rangle$$

with the tag "Tag" can be represented as a PROLOG term $\mathsf{Tag} : \mathsf{As} : \mathsf{C}$, where As is an association list for the attribute/value–pairs $a_i = "v_i"$ and C represents the contents, i.e., the subelements. E.g., for the XML representation of the atom `tc(U1, U2)` we get:

```
atom:[predicate:'(user:tc)/2']:[
   var:[name:'U1']:[], var:[name:'U2']:[] ]
```

There exist several possibilities to *access* and *update* an object `O` in field notation using a binary infix predicate "`:=`", which evaluates its right argument and tries to unify the result with its left argument. Given an element tag `E` and an attribute `A`, we use the call `X := O^E` to select the `E`–subelement `X` of `O`, and we use `Y := O@A` to select the `A`-value `Y` of `O`; the application of selectors can be iterated, cf. path expressions in XML query languages [1]. On *backtracking* all solutions to a path expression can be obtained.

```
?- Atom = atom:[predicate:'(user:tc)/2']:[
      var:[name:'U1']:[], var:[name:'U2']:[] ]
   P := Atom@predicate, V := Atom^var,
   findall( N,
      N := Atom^var@name,
      Ns ).

P = '(user:tc)/2', V = var:[name:'U1']:[], Ns = ['U1', 'U2']

Yes
```

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

```
?- Atom = atom:[predicate:'(user:tc)/2']:[
      var:[name:'U1']:[], var:[name:'U2']:[] ],
   Atom_2 := Atom*[@predicate:'closure/2'].

Atom_2 = atom:[predicate:'closure/2']:[
   var:[name:'U1']:[], var:[name:'U2']:[] ]

Yes
```

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 subcomponents according to substitution rules in the style of XSLT, and the manipulation of path or tree expressions.

3 Visualization of PROLOG Rules in VISUR

For visualizing the call structure of rule–based systems the concept of *dependency graphs*, which is well–known from deductive databases [2], is used. DATALOG programs can be analysed using diverse dependency graphs, e.g., the *rule/goal graph* and the *goal graph*.

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.

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 $G_r^{\mathrm{rg}} = \langle V_r^{\mathrm{rg}}, E_r^{\mathrm{rg}} \rangle$ of r is well–known from literature:

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

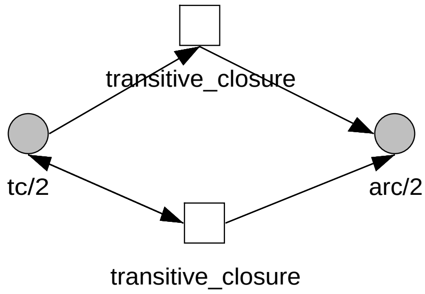

Figure 1: Rule/Goal Graph in VISUR

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

Meta–Call Predicates in PROLOG. Unfortunately, these graphs cannot handle practical applications of PROLOG properly, since they do not take into account meta–call predicates. Meta–call predicates allow for higher–order programming; some well–known meta–call predicates are for instance `call/1`, `findall/3`, `forall/2`, `maplist/3`, and `checklist/2`. E.g., the evaluation of `findall/3` within the following rule repeatedly calls the predicate `calls_uu_pred/2` and collects the results in the list `Predicates`:

```
calls_uu(Program, (Unit_1-Unit_2):N) :-
   unit(Unit_1), unit(Unit_2),
   Unit_1 =\= Unit_2,
   findall( Predicate,
      calls_uu_pred(Program, (Unit_1-Unit_2):Predicate),
      Predicates ),
   length(Predicates, N).
```

To treat *meta–call predicates* in PROLOG programs adequately, we will define extended dependency graphs, which include the predicates called in the parameter list of a meta–call predicate. Figure 2 contains the extended dependency graph of the rule generated by VISUR. We use a *rhombus* as the symbol for meta–call predicates in order to distinguish them from ordinary predicates, and we use a *triangle* as the symbol for built–in predicates.

In the rule/goal graph there would be no edge from `findall/3` to `calls_uu_pred/2`. We introduce the notation $p_B \prec p_A$ for expressing that an atom B with the predicate symbol p_B is called within an atom A with the predicate symbol p_A. Thus, in our example we get

`calls_uu_pred/2` $\prec$ `findall/3`.

In VISUR it is possible to define the relation $\prec$ customized to the individual user needs. E.g., if the predicate symbol p_B is constructed at runtime by appending some dynamic suffix p''_B to a static prefix p'_B, i.e. $p_B = p'_B \circ p''_B$, then we can specify that $p_B \prec p_A$ holds for all p_B with the prefix p'_B. In Section 4 we will see such predicates, where $p'_B = \mathsf{calls\_}$ and $p''_B \in \{\, \mathsf{fp}, \mathsf{uu} \,\}$.

The Extended Rule/Goal Graph. The extended rule/goal graph $G_P^{\text{erg}} = \langle V_P^{\text{erg}}, E_P^{\text{erg}} \rangle$ takes care of the fact that an atom B can be called within another atom A. The node set $V_P^{\text{erg}} = \cup_{A \in \mathit{HB}_P} V_A^{\prec}$ and the edge set $E_P^{\text{erg}} = \cup_{A \in \mathit{HB}_P} E_A^{\prec}$ are obtained by collecting the following node and edge sets for the ground atoms A in the Herbrand base HB_P:

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

The extended rule/goal graph of Figure 2 visualizes most of the PROLOG rules of Section 4; the rule for the predicate `calls_uu/2`, that is shown above, is one of them.

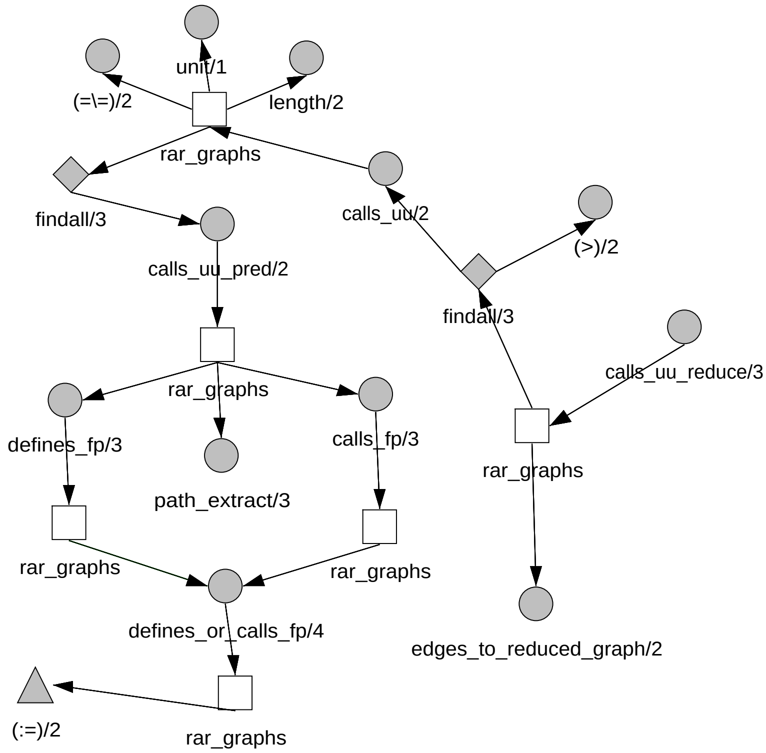

Figure 2: Extended Rule/Goal Graph in VISUR

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

4 Analysis of Dependency Graphs in RAR

In this section we present some PROLOG rules for extracting structural design information from *hierarchically structured* PROLOG programs. We will demonstrate by some case studies how the quality of the arrangement of the predicates in the source code files can be measured.

We have applied our analysis to the system DISLOG, which is hierarchically structured into units, modules, and files; i.e., in DISLOG a module consists of serveral files, whereas in PROLOG a module usually consists of a single file. Currently, the complete name (path) of a DISLOG source file is of the form

```
DisLog/sources/<unit>/<module>/<file>,
```

and the call `path_extract(Type, Path, X)` extracts `X = <unit>`, `X = <module>`, and `X = <file>`, from such a path `Path`, where `Type` is one of `unit`, `module`, and `file`, respectively.

4.1 Basic Dependencies between Files and Predicates

Given a program `Program` in field notation, the following predicate `defines_fp/3` describes that the predicate `Predicate` is defined (i.e., it occurs in the head of a rule) in the file `File`, and `calls_fp/3` describes that `Predicate` is called in the body of a rule in `File`:

```
defines_fp(Program, File, Predicate) :-
   defines_or_calls_fp(Program, File, head, Predicate).

calls_fp(Program, File, Predicate) :-
   defines_or_calls_fp(Program, File, body, Predicate).

defines_or_calls_fp(Program, File, Selector, Predicate) :-
   Rule := Program^definition^rule,
   File := Rule@file,
   Predicate := Rule^Selector^atom@predicate.
```

The predicate `defines_or_calls_fp/4`, which extracts the common part of the two predicates `defines_fp/3` and `calls_fp/3`, uses the variable `Selector` in a path expression. E.g., calling `defines_fp(Program, File, '(user:calls_fp)/3')` for the representation of the DISLOG system determines the file that defines `'(user:calls_fp)/3'`:

```
File = 'DisLog/sources/databases/rar/rar_metrics'
```

4.2 Derived Dependencies between Units

A unit `Unit_1` calls a unit `Unit_2`, if there exists a file `File_2` in `Unit_2` defining a predicate `Predicate` that is called in a file `File_1` in `Unit_1`. In practice it turned out that in DISLOG there exist calls between most units, but there are great differences between the numbers of such calls. The following predicate `calls_uu/2` returns elements of the form `(Unit_1-Unit_2):N` by counting the number `N` of predicate calls from `Unit_1` to `Unit_2`:

```
calls_uu(Program, (Unit_1-Unit_2):N) :-
   unit(Unit_1),
   unit(Unit_2),
   Unit_1 =\= Unit_2,
   findall( Predicate,
      calls_uu_pred(Program, (Unit_1-Unit_2):Predicate),
      Predicates ),
   length(Predicates, N).
```

```
calls_uu_pred(Program, (Unit_1-Unit_2):Predicate) :-
   calls_fp(Program, File_1, Predicate),
   defines_fp(Program, File_2, Predicate),
   path_extract(unit, File_1, Unit_1),
   path_extract(unit, File_2, Unit_2).
```

In most situations where a unit calls another unit very few times, these calls are due to missplaced predicate definitions; these predicate definitions could be moved to other places, such that the calls disappear. In Figure 3 the remaining part of the unit dependency graph of DISLOG is shown.

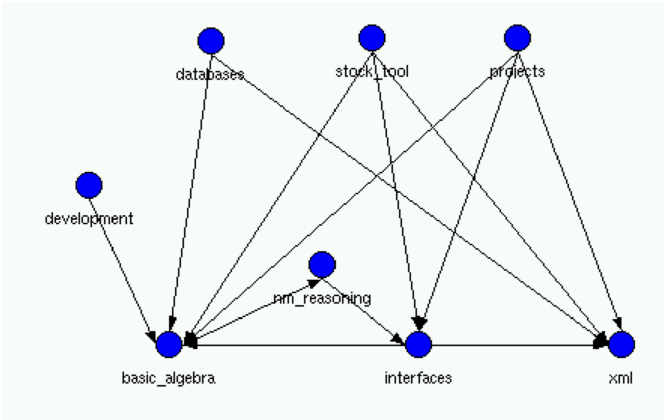

Figure 3: Unit Dependency Graph of DISLOG in VISUR

Figure 3 suggests to partition the set of 8 DISLOG units into two *levels*: The lower level consists of the units `basic_algebra`, `interfaces`, `nm_reasoning`, and `xml`; the higher level consists of `development`, `databases`, `stock_tool`, and `projects`.

We have computed the *reduction* of the unit dependency graph of Figure 3 to *strongly connected components* based on the standard library `ugraphs.pl` of SWI–PROLOG: `[basic_algebra, interfaces, nm_reasoning]` is a strongly connected component. By joining the strongly connected components into higher–level packages, we can obtain an *acyclic* package structure.

Extracting a Maximal Acyclic Unit Structure

Another alternative for *improving the design* of a system could be to identify a small set of predicates that should be moved from one unit to another unit, such that the resulting unit dependency graph is acyclic. This could be done by extracting a maximal acyclic subgraph of the unit dependency graph, such that the neglected edges represent the smallest number of calls. Thus, the following program derives a relation `arc/2` that is a subset of the relation `calls_uu/2`.

```
arc(U1, U2) :-
   calls_uu(U1, U2),
   not(tc(U2, U1)).

tc(U1, U2) :-
   arc(U1, U2).
```

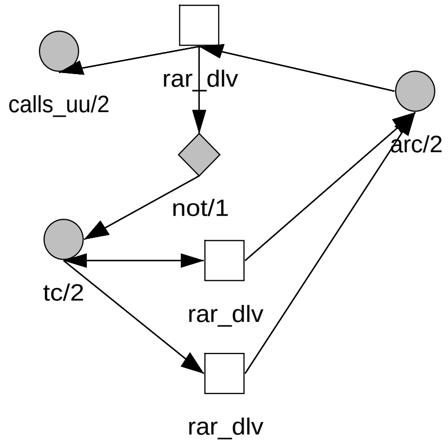

Figure 4: Rule/Goal Graph in VISUR

```
tc(U1, U2) :-
   arc(U1, U3), tc(U3, U2).
```

The program cannot be evaluated using PROLOG, since the recursion defining the predicate `arc/2` envolves negation, cf. Figure 4. Thus, we have used the *disjunctive* logic programming system dlv [4] for evaluating the program. dlv is able to derive the so–called *answer sets*, which are certain minimal sets of atoms satisfying the rules. Obviously, we can focus on the non–trivial strongly connected component `[basic_algebra, interfaces, nm_reasoning]`. The (partial) solutions obtained by *answer set programming* with dlv are given in Figure 5. They show that we can obtain three different maximal acyclic unit structures.

The most reasonable one is given by the answer set M_3, which indicates to neglect the edge from `basic_algebra` to `nm_reasoning` in the unit dependency graph; in that case only 20 predicate defintions would have to be moved from one unit to another unit (obviously, this might not always be possible). In comparison, M_1 would neglect 187 calls, and M_2 would neglect 191 calls.

Unit	Unit	Calls	M_1	M_2	M_3
basic_algebra	nm_reasoning	20	✓	✓	
interfaces	basic_algebra	41		✓	✓
nm_reasoning	interfaces	45	✓		✓
nm_reasoning	basic_algebra	146			✓

Figure 5: Answer Sets Computed by dlv

We have used dlv since we are planning to analyse *disjunctive* logic programs in the future as well. Our current program above, however, is a *normal* logic program, and we could also evaluate it using the answer set programming system Smodels, which is restricted to normal logic programs. Both dlv and Smodels can compute optimal models; thus, it is possible to compute the *optimal* answer set M_3 without computing all answer sets.

5 Analysis of the Internal Structure of Rules

In this section we analyze the internal structure of individual PROLOG rules or definitions. In particular we compute *software metrics*, we search for *design patterns*, and we apply *refactoring* techniques. The examples demonstrate the usefulness of our PROLOG library for handling XML representations of source code.

As an example of PROLOG–specific software metrics, in Section 5.1 we show how the percentage of directly recursive predicates in DISLOG can be calculated, and we compare it with the percentage of meta–call predicates.

In Section 5.2 we describe the detection and the refactoring of patterns. We detect directly recursive predicates having a special form that could in fact be replaced using the meta–call predicate `maplist/3`. Other patterns that one could look for are the accumulator pattern for traversing a list – as it can be found in the predicate `sumlist/2` of the library `lists.pl` of SWI–PROLOG.

5.1 Directly Recursive Rules vs. Meta–Call Predicates

The following predicate counts the number of directly recursive rules for each unit. The results are a multiset `Units` consisting of pairs `Unit:K`, such that `K` is the number of directly recursive rules in `Unit`, and an integer `N`, which is the overall number of directly recursive rules in all units.

```
directly_recursive_predicates(Program, Units, N) :-
   findall( Unit,
      ( Definition := Program^definition,
        Predicate := Definition@predicate,
        Rule := Definition^rule,
        Predicate := Rule^body^atom@predicate,
        Path := Rule@file,
        path_to_unit_name(Path, Unit) ),
      Units_2 ),
   list_to_multiset(Units_2, Units),
   length(Units_2, N).
```

The predicate `list_to_multiset/2` converts a list – with possibly multiply occurring elements – into a list of pairs `Element:Multiplicity`, which represents a multiset. Applying the predicate `directly_recursive_predicates/3` to DISLOG has shown that there are 1645 directly recursive rules in DISLOG. Figure 6 shows the percentage of such predicates per module.

Sometimes, meta–call predicates, such as `maplist/3` or `checklist/2` can be used instead of directly recursive calls. In many cases they are a better choice, since they are more readable. Moreover, meta–call predicates are often implemented in the kernel, and they are faster than directly recursive calls.

The distribution in Figure 6 shows, that in the units of level 1 often directly recursive calls are used, whereas the distribution in Figure 7 shows, that in the units of level 2 meta–call predicates are used more often; the only exception is the unit `xml`.

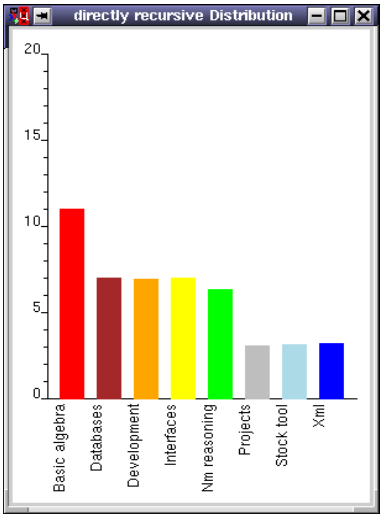

Figure 6: Rules with Directly Recursive Predicates (in %)

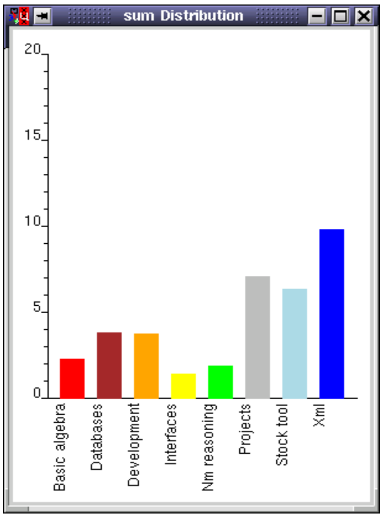

Figure 7: Rules with Meta–Call Predicates (in %)

5.2 Detection of Patterns and Refactoring

The following two rules define the predicate `vector_multiply/3` using direct recursion:

```
vector_multiply(F, [X|Xs], [Y|Ys]) :-
   Y is F * X,
   vector_multiply(F, Xs, Ys).
vector_multiply(_,[],[]).
```

Every list element `X` of the input list is multiplied by `F` to obtain a list element `Y` of the output list. A more abstract – and more elegant – implementation of `vector_multiply/3` can be obtained using the meta–call predicate `maplist/3`:

```
vector_multiply(F, Xs, Ys) :-
   maplist( multiply(F),
      Xs, Ys ).

multiply(F, X, Y) :-
   Y is F * X.
```

The version using `maplist/3` is shorter, since the predicate `multiply/3` already exists in DISLOG. For complex transformations – where the input element `X` is transformed into the output element `Y` using a complex sequence of operations – the version using `maplist/3` in most cases is much more readable. The same holds for predicates with many parametric arguments such as `F` in `vector_multiply/3`.

An alternative implementation is possible using the PROLOG library loops.pl, cf. [14], where it is argued that the enhancement of PROLOG by concepts – such as logical loops – that are familiar from other programming languages increases productivity and mantainability:

```
vector_multiply(F, Xs, Ys) :-
   ( foreach(X, Xs), foreach(Y, Ys) do
         Y is F * X ).
```

Thus, we have implemented an automatic detection and refactoring of maplist patterns. The following two important predicates are applied to a linear recursive rule for the predicate symbol `Predicate`. `list_argument/4` is applied to the head atom `Atom` of the rule; it detects a list pattern `X:Xs` and returns its argument position `N`. `variable_argument/4` is applied to a body atom `Atom` of the rule; it checks if the argument at position `N` is `Variable`:

```
list_argument(Atom, Predicate, N, X:Xs) :-
   Predicate := Atom@predicate,
   Argument := Atom-nth(N)^_,
   '.' := Argument@functor,
   X := Argument-nth(1)^var@name,
   Xs := Argument-nth(2)^var@name.

variable_argument(Atom, Predicate, N, Variable) :-
   Predicate := Atom@predicate,
   Argument := Atom-nth(N)^_,
   Variable := Argument@name.
```

`Argument := Atom-nth(N)^_` selects the `N`–th argument of `Atom`, independently of its tag, and, if `Argument` is a list (i.e., its functor is "`.`"), then `X := Argument-nth(1)^var@name` selects the head `X` of the list and `Xs := Argument-nth(2)^var@name` selects the tail `Xs`. For our recursive rule we can apply these predicates to the head and the body atom, respectively, with the predicate `vector_multiply/2`:

```
?- Atom_1 = atom:[predicate:'vector_multiply/3']:[
      var:[name:'F']:[],
      term:[functor:'.']:[
```

```
         var:[name:'X']:[], var:[name:'Xs']:[]],
      term:[functor:'.']:[
         var:[name:'Y']:[], var:[name:'Ys']:[]]],
   Atom_2 = atom:[predicate:'vector_multiply/3']:[
      var:[name:'F']:[],
      var:[name:'Xs']:[], var:[name:'Ys']:[]],
   list_argument(Atom_1, P, N, X:Xs),
   variable_argument(Atom_2, P, N, Xs).

P = 'vector_multiply/3', X = 'X', Xs = 'Xs', N = 2;
P = 'vector_multiply/3', X = 'Y', Xs = 'Ys', N = 3

Yes
```

If we find a pair `N,N'` of corresponding positions (above we found `N = 2` and `N' = 3`), then further predicates have to check that there is no other recursive rule for `Predicate`, that the list variables `Xs` and `Ys` don't occur in the remaining body atoms (i.e., in "`Y is F * X`" in our example), that the arguments in the other positions in the recursive atom in the head and the body, respectively, are identical (i.e., the first argument `F`), and that the non–recursive rule for `Predicate` is just an atom with "`[]`" in the positions `N,N'` and underscore variables in the remaining positions.

It turned out that there exist 176 maplist patterns in the 4 basic level units of DISLOG; on the other hand, we have computed that there are 78 calls to `maplist/3` in these 4 units. In the 4 higher–level units, there exist 54 maplist patterns, and there are 388 calls to `maplist/3`.

6 Conclusions

VISUR/RAR can be used for analyzing and for refactoring source code; it can help to improve the initial design of a system. By using an extended definition of dependency graphs it became possible to treat meta–call predicates in PROLOG adequately. Our approach is based on heuristics; in general it is not statically decidable which calls are to be actually done at run time in the presence of meta–calls and rules that are asserted or retracted at runtime.

The integration of XML processing, visualization, and reasoning in the logic programming environment XPCE–PROLOG has created a powerful and flexible tool. The library FNQUERY, which we are using for accessing the components of the XML representations of the source code, has also been used in other projects. For example, in [8] it is shown how mathematical knowledge in XML can be managed nicely using FNQUERY.

The predicates for analyzing dependency graphs of PROLOG programs of Section 4 can easily be adapted to JAVA programs by redefining the basic predicates `defines_fp/3` and `calls_fp/3`, and by considering classes instead of files and methods instead of predicates. In [10] we defined such predicates based on an XML representation of JAVA source code; in that paper we have also investigated some simple predicates for analyzing dependency graphs of PROLOG source code, which are different from the predicates described in Section 4 of the present paper.

In the future, we will gradually extend VISUR/RAR with additional features. We intend to implement sophisticated methods for *program analysis* from *software engineering* [3, 5, 6, 16], and we want to integrate further *refactoring techniques* for PROLOG and for JAVA code, which have been developed in [18] for PROLOG.

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] *S. Diehl (Ed.):* Software Visualization: International Seminar, Dagstuhl Castle, Germany, Springer LNCS 2269, 2002.

[4] *T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello:* A Deductive System for Non–Monotonic Reasoning, Proc. Fourth Intl. Conf. on Logic Programming and Non–Monotonic Reasoning LPNMR 1997, Springer LNAI 1265, 1997.

[5] *H. Erdogmus, O. Tanir (Eds.):* Advances in Software Engineering - Comprehension, Evaluation, and Evolution, Springer, 2002.

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

[7] *M. Hanus, J. Koj:* An Integrated Development Environment for Declarative Multi–Paradigm Programming, Proc. Workshop on Logic Programming Environments WLPE 2001.

[8] *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 2594.

[9] *M. Hopfner, D. Seipel:* Reasoning about Rules in Deductive Databases, Proc. 17th Workshop on Logic Programming WLP 2002.

[10] *M. Hopfner, D. Seipel, J. Wolff von Gudenberg:* Comprehending and Visualising Software based on XML Representations and Call Graphs, Proc. 11th IEEE International Workshop on Program Comprehension IWPC 2003.

[11] *R. Holt, A. Winter, A. Schürr:* GXL: Towards a Standard Exchange Format, Proc. Working Conference on Reverse Engineering WCRE 2000, http://www.gupro.de/GXL/

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

[13] *M. Kifer, G. Lausen:* F–Logic: A Higher–Order Language for Reasoning about Objects, Proc. ACM SIGMOD Conference on Management of Data, 1989.

[14] *J. Schimpf:* Logical Loops, Proc. Intl. Conference on Logic Programming ICLP 2002.

[15] *A. Serebrenik, B. Demoen:* Refactoring Logic Programs, Proc. Intl. Conference on Logic Programming ICLP 2003 (Poster Session).

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

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

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

[19] *G. Wagner:* How to Design a General Rule Markup Language, Proc. Intl. Workshop on XML Technologies for the Semantic Web, XSW 2002.

[20] *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/
J. Wielemaker, A. Anjewierden: Programming in XPCE/PROLOG http://www.swi-prolog.org/

Appendix

We are representing PROLOG programs in XML according to the following DTD, which is suitable for handling disjunctive logic programs with arbitrarily many head atoms:

```
<!ELEMENT program (definition*)>
<!ELEMENT definition (rule*)>
<!ELEMENT rule (head, body)>
<!ELEMENT head (atom*)>
<!ELEMENT body (atom*)>
<!ELEMENT atom ((term|var)*)>
<!ELEMENT term (term*)>

<!ATTLIST definition predicate CDATA #required>
<!ATTLIST rule file CDATA #required>
<!ATTLIST atom predicate CDATA #required>
<!ATTLIST term functor CDATA #implied>
<!ATTLIST var name CDATA #implied>
```

The arguments of an atom are either terms or variables; constants are represented as terms without subterms, where the constant is stored in the attribute `functor`.