Comprehending and Visualizing Software Based on XML–Representations and Call Graphs

Marbod Hopfner[1], Dietmar Seipel[2], and 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

Overview of VISUR/RAR

For many procedural programming languages there exist powerful *Integrated Development Environments (IDEs)*. such as IBM's ECLIPSE for JAVA, and Together for JAVA, C++, Visual Basic, etc., which are widely used. Also in the logic programming community there are tools for comfortably programming and for analyzing source code, such as the IDE for XPCE/SWI–PROLOG and the tool CIDER for the functional–logic language CURRY.

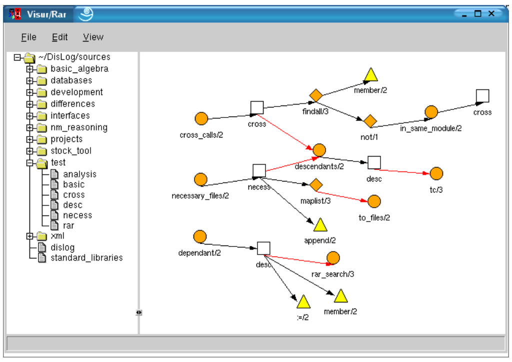

Figure 1. Graphical User Interface of VISUR

We have implemented a package VISUR/RAR in XPCE/SWI–PROLOG, which provides some further functionality of an IDE: it allows for the *visualization* of rules (VISUR) together with the *reasoning* about rule structures (RAR). VISUR/RAR is part of the toolbox DISLOG [8], which is also developed under XPCE/SWI–PROLOG; the functionality of DISLOG ranges from reasoning in disjunctive deductive databases to applications such as the management and visualization of stock information.

The deductive database of RAR manages rules for analyzing PROLOG–code based on suitable *dependency graphs* and rules for recovering the design of PROLOG– and JAVA–software using a query language FNQUERY for XML–representations, which we have developed. It can be applied for improving the design of rule–based systems, for computing certain *software metrics*, and for supporting *refactoring techniques* [6]. The obtained results can be visualized using graphs and tables in VISUR.

Analysis of Software

The system VISUR/RAR supports developers in the comprehension and the further development of software projects in PROLOG or JAVA. VISUR makes it easy for the user to become acquainted with the source code by *visualizing* coherences between different source files of a project or by visualizing rule calls. Using RAR it is possible to *analyze* source code customized to the individual needs of a user – and to visualize the results graphically or in tables with VISUR. For example it is possible to answer the following questions using RAR:

- Which parts (files, modules) of a project are needed in order to make a specific predicate work correctly ?
- Which predicates call predicates from other modules, and which of them transitively call themselves, i.e., are recursive ?
- Which relationships exist between classes and methods in JAVA–source code ?

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. In principle, VISUR/RAR could be applied to various kinds of rules in expert systems.

We have also applied RAR for analyzing JAVA–code. We use XML–representations such as JAML [5], JavaML [2], and srcML [4], for representing the JAVA–code and a library called FNQUERY for processing the JAML–documents.

Call Graphs for Rule Systems

In the field of logic programming and deductive databases *dependency graphs* are used for visualizing the call structure of rule systems, cf.[3].

Extended Rule/Goal Graphs. We have defined *extended dependency graphs*, which also take into account the predicates called within the parameter list of meta–predicates. Figure 1 shows the extended rule/goal graph of an RAR–program that is completely described in [7]; the rule for the predicate `cross_calls/2` is given below. We use a *diamond* as the symbol for meta–predicates, such as `findall/3` and `maplist/3`, and rule nodes are labelled by module names, e.g. by `cross`.

Module Dependency Graphs. The *module dependency graph* shows the calls across modules. E.g., the module dependency graph of Figure 2 shows the hierarchical structure of our RAR–program. 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.

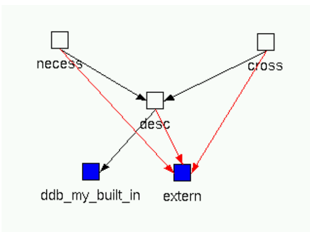

Figure 2. Module Dependency Graph

Example (Calls Across Modules). The following predicate `cross_calls/2` determines the list `Ds` of all predicates which are transitively called by a given predicate `P` – this could be done based on the extended rule/goal graph – and then – using the meta–predicate `findall/3` – it selects the list `CCs` of all predicates which are not in the same module as `P`.

```
cross_calls(P, CCs) :-
  descendants(P, Ds),
  findall( Pred,
    ( member(Pred, Ds),
      not(in_same_module(P, Pred)) ),
    CCs ).

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

In the table of Figure 3, which was also generated by VISUR, the *calls across modules* in our PROLOG–program are counted: from each of the modules `cross` and `necess`, respectively, there is one call to the module `desc`.

module_1	module_2	calls
cross	desc	1
cross	necess	0
desc	cross	0
desc	necess	0
necess	cross	0
necess	desc	1

Figure 3. Calls Accross Modules

References

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

[2] *G.J. Badros:* JavaML: A Markup Language for Java Source Code, 9th International World Wide Web Conference, 2000.

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

[4] *M.L. Collard, J.I. Maletic, A. Marcus:* Supporting Document and Data Views of Source Code, Proc. 2nd ACM Symposium on Document Engineering DocEng 2002.

[5] *G. Fischer, J. Wolff von Gudenberg:* JAML – An XML–Representation of JAVA–Source Code, Technical Report, University of Würzburg, 2003.

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

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

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